计算机程序设计语言系列丛书

DOS 6.22

技术手册

——汇编语言教程

MASM 6.11

旗標

希望

学苑出版社

施威铭 著

台湾旗标出版有限公司新到中文简体字版电脑图书

序号	购书编号	书　　名	定价
1	1306	PC Tools for Windows 使用指南	39.00 元
2	1359	TSR 程序设计专题研究	25.00 元
3	1366	TSR 常驻程序原理与实例选辑	34.00 元
4	1387	电脑病毒彻底研究	19.00 元
5		Animator Pro 动画制作系统实例手册	35.00 元
6	1426	PC 图像处理技巧与实践	56.00 元
7		Windows 程序设计实务(一)基础篇	38.00 元
8	1810	MS-DOS 6.22 参考手册	23.00 元
9	1812	MS-DOS 6.22 使用手册	25.80 元

欲购以上图书的朋友请与 010－2541992，2562329 书刊部联系，

或传真 010－2579874，2561057 书刊部

ISBN7—5077—0807—1/TP · 18

本册定价:18.50 元

计算机程序设计语言系列丛书

DOS 6.22 技术手册——汇编语言教程

施威铭 编著

麻信洛 徐文军 改编

燕卫华 审校

学 苑 出 版 社

(京)新登字 151 号

内容提要

本书从最基本的原理开始，对 80x86 CPU 的结构、PC 的组成、80x86 的指令、DEBUG 的用法以及用 MASM 6.11 开发汇编语言程序一一给予详细说明。全书列出了大量的例子，供使用者逐一印证，以帮助读者克服学习汇编语言的困难。本书的重点在于为读者建立编写汇编语言程序的基本概念，因此所举的例子注重对过程的了解，而非例子本身的实用性。本书适用于计算机操作人员及开发人员。

需要本书的读者，请直接与北京海淀 8721 信箱书刊部联系，电话：2562329，邮编：100080。

在本书的改编过程中，刘杰做了大部分的录入工作，李晓中、张景生、宋涛等在校对和审核的过程中做了大量的工作，在此一并致以衷心的感谢。

版权声明

本书繁体字中文版原书名为《MASM 6.11 组合语言实务》，由旗标公司出版，版权归旗标公司所有。本书简体字中文版由旗标公司授权出版。未经出版者书面许可，本书的任何部分不得以任何形式或任何手段复制或传播。

计算机程序设计语言系列丛书

DOS 6.22 技术手册——汇编语言教程

编　　著：施威铭
改　　编：麻信洛　徐文军
审　　校：燕卫华
责任编辑：甄国宪
出版发行：学苑出版社　**邮政编码**：100036
社　　址：北京市海淀区万寿路西街 11 号
印　　刷：　施园印刷厂
开　　本：787×1092　1/16
印　　张：12.875　**字数**：295 千字
印　　数：1～5000 册
版　　次：1995 年 9 月北京第 1 版第 1 次
ISBN7－5077－0807－1/TP·18
本册定价：18.50 元

学苑版图书印、装错误可随时退换

目 录

第一篇 基本概念

第一章 数字系统……1
1.1 汇编语言与数字系统的关系……1
1.2 二进制、十进制和十六进制数字系统……2
1.3 二进制数、十进制数、十六进制数之间的转换关系……4
1.4 二进制数与十六进制数的四则运算……7
1.5 正负数的表示法……10
1.6 电脑数据的基本单位:位(Bit)、字节(Byte)、字(Word)……11
1.7 两种常见的符号系统:ASCII 码和汉字编码……13
第二章 PC 的基本构造……16
2.1 PC 的结构简介……16
2.2 80x86 CPU 及其寄存器组……16
2.3 存储器的结构……26
2.4 80x86的分段式存储器管理……29
2.5 80x86指令的寻址方式……35

第二篇 必备的工具

第三章 如何使用调试程序 DEBUG……43
3.1 进入及退出 DEBUG……43
3.2 计算十六进制数……44
3.3 查看存储器的内容……44
3.4 将数据输入存储器……45
3.5 中断原理的说明……47
3.6 用 DEBUG 编写汇编语言程序……48
3.7 程序的存盘与载入……54
3.8 执行程序……56
3.9 跟踪程序……57
第四章 汇编语言的开发流程和程序结构……60
4.1 汇编语言的程序开发流程……60
4.2 汇编语言的语法……61
4.3 常用的 MASM 伪指令……65
4.4 MASM 的程序结构……69

4.5　简单的程序范例 ………………………………………………………… 70
4.6　使用 ML.EXE、LINK.EXE 来汇编和连接程序 …………………………… 70
4.7　.COM 文件和.EXE 文件 ………………………………………………… 72

第三篇　指令介绍与练习

第五章　数据传送指令集 ………………………………………………… 77
5.1　数据传送指令 ………………………………………………………… 77
5.2　堆栈存取指令 ………………………………………………………… 81
5.3　地址操作指令 ………………………………………………………… 83
5.4　表操作指令 …………………………………………………………… 86
5.5　程序举例 ……………………………………………………………… 87
第六章　算术运算指令集 ………………………………………………… 91
6.1　进位与溢出的原理 …………………………………………………… 91
6.2　加法与减法指令 ……………………………………………………… 92
6.3　乘法指令 ……………………………………………………………… 97
6.4　除法指令 ……………………………………………………………… 99
6.5　有符号数扩充指令 …………………………………………………… 100
6.6　BCD 运算调整指令 …………………………………………………… 102
6.7　程序举例 ……………………………………………………………… 108
第七章　位运算指令集 …………………………………………………… 111
7.1　逻辑运算指令 ………………………………………………………… 111
7.2　移位指令 ……………………………………………………………… 114
7.3　循环移位指令 ………………………………………………………… 117
7.4　程序举例 ……………………………………………………………… 119
第八章　程序流程控制指令集 …………………………………………… 123
8.1　标志设定指令 ………………………………………………………… 123
8.2　比较指令 ……………………………………………………………… 125
8.3　跳转指令 ……………………………………………………………… 127
8.4　循环指令 ……………………………………………………………… 130
8.5　有关子程序的指令 …………………………………………………… 132
8.6　中断指令 ……………………………………………………………… 134
8.7　程序举例 ……………………………………………………………… 136
第九章　程序流程控制伪指令 …………………………………………… 140
9.1　IF、ELSEIF、ELSE、ENDIF 伪指令 ……………………………………… 140
9.2　WHILE、ENDW、REPEAT、UNTIL、BREAK、CONTINUE 伪指令 ………… 144
9.3　程序举例 ……………………………………………………………… 147
第十章　字符串操作指令 ………………………………………………… 151
10.1　字符串操作指令总览 ……………………………………………… 151

10.2 字符串传送指令 …… 151
10.3 字符串比较指令 …… 153
10.4 字符串搜索指令 …… 155
10.5 字符串的载入与存储指令 …… 157
10.6 重复前缀指令 …… 159
10.7 程序举例 …… 161

第四篇 高级应用

第十一章 子程序的使用 …… 165
11.1 如何定义子程序 …… 165
11.2 近程调用和远程调用 …… 166
11.3 内部调用与外部调用 …… 167
11.4 PUBLIC 与 EXTRN 的使用 …… 171
11.5 程序举例 …… 173
11.6 子程序的再利用 (Reuse) …… 175
11.7 程序库 …… 176
11.8 LIB 的使用技巧 …… 178
第十二章 宏(MACRO)的使用 …… 181
12.1 何谓宏 …… 181
12.2 宏与子程序的比较 …… 183
12.3 INCLUDE 伪指令 …… 185
12.4 宏的参数传递 …… 186
12.5 宏内的标号与 LOCAL 定义 …… 188
12.6 程序举例 …… 189
附录 A ML.EXE 的命令参数 …… 194
附录 B MASM 6.11的系统需求与安装程序 …… 196

第一篇　基本概念

第一章　数字系统

计算机是一种机器，机器所使用的语言和我们人类的语言大不相同。所以，在接触计算机之初，首先要做的事就是了解计算机的语言——数字符号系统。

本章所要介绍的内容有：

□二进制、十进制、十六进制数字系统

□二进制、十进制、十六进制之间的转换及其四则运算

□正负数的表示法

□数据的基本单位：位(Bit)、字节(Byte)和字(Word)

最后我们会说明两种常用的符号系统：ASCII码以及汉字编码。这是我们学习汇编语言的准备工作。让我们以愉快的心情来学习吧！

汇编语言是计算机的基本程序设计语言，速度快、效率高、但不易为人所了解。事实上，学习汇编语言就像学习外语一样，不但要学习其文字、语法，还要认识其生活环境、法令、制度，相对于学习汇编语言来说，就是要认识电脑的系统环境。因此，要想学好汇编语言，就必须对电脑内部的数制、符号、结构……等有相当全面的了解。所以，在本书的首章，我们将介绍与电脑系统紧密相关的数字系统。有了这些基本认识，我们才可以顺利的进入电脑世界。当然，勤于练习是学习任何语言的不二法门，这是研读本书随时要牢记在心的。

1.1　汇编语言与数字系统的关系

电脑内部的基本构造是开关装置。所谓开关装置是指以高电位(ON)来表示"1"；以低电位(OFF)来表示"0"，而电脑所能接受的命令就是一连串"0"和"1"的组合，例如："0110110101110011"。因此在早期的电脑上，程序设计人员只能以机器语言给电脑下命令，才能使电脑进行运算(早期的程序设计甚至是通过更改电路来实现的)。以下面这行二进制的机器语言为例，我们分别将它转换成十六进制的机器码或汇编语言的格式：

```
00000101 00101010 00000000        ;二进制机器码
05 2A 00                          ;十六进制机器码
ADD AX,42                         ;汇编语言
```

如果我们对于上述三种情况都不熟悉，显然十六进制机器码比二进制机器码更容易记忆，而当我们知道了在汇编语言中ADD这个指令是代表相加的意思时，汇编语言又比十六进制机器码更容易令人接受，这也是程序开发者用汇编语言取代机器语言的原因。可见，我们之所以需要不同的数字系统，只是为了更方便地分析电脑工作的原理！

从上述的例子可以看出，电脑内部的数据适合以二进制来描述。但由于人们不习惯于使用1000101011010110这种一长串的数字表示法，因此才折衷采用了十六进制。在使用汇编语言的过程中，我们并不总是固定使用二进制、十进制或十六进制，而是根据具体的情况来选用某

种数制。这就好比一些原文的科技名词，在中文中还没找到一个比较贴切的的用语来表达它时，就直接将其原文穿插在中文中，这样做往往更容易使人理解。反复使用不同的数字系统也是这个道理！

1.2 二进制、十进制和十六进制数字系统

1.2.1 二进制数字系统

前面我们已经提到了：二进制是最适合于描述电脑内部运算的数字系统。所以我们首先必须知道二进制是怎样处理的。以我们所熟悉的十进制数字系统来说，其含义是指在这种数字系统中，计数是逢 10 进位的。同理，二进制数字系统，就是逢 2 进位的数字系统，两者之间的对应关系如表 1.1 所示。

表1.1 二进制数与十进制数对照表

		二进制数	十进制数	
		0 = 0	0	加10次1才进位
		0+1 = 1	1	
每加两次1就进位	进位 →	1+1 = 10	2	
		10+1 = 11	3	
	进位 →	11+1 = 100	4	
		100+1 = 101	5	
	进位 →	101+1 = 110	6	
		110+1 = 111	7	
	进位 →	111+1 = 1000	8	
		1000+1 = 1001	9	
	进位 →	1001+1 = 1010	10	
		⋮	⋮	

1.2.2 十六进制数字系统

十六进制数字系统就是逢 16 进位的数字系统。也就是由 0、1、2、3、4、5、6、7、8、9、A、B、C、D、E、F 所组成的，其中 A、B、C、D、E、F 六个符号相当于十进制数中的 10、11、12、13、14、15 六个数字。两者对照见表 1.2 所示。

表1.2　十六进制数与十进制数对照表

	十六进制数	十进制数
	0	0
	0+1= 1	1
	1+1= 2	2
	⋮	⋮
	8+1= 9	9
	9+1= A	10
	A+1= B	11
	B+1= C	12
	C+1= D	13
	D+1= E	14
	E+1= F	15
进位 →	F+1=10	16
	10+1=11	17
	⋮	⋮

表1.3　二进制数、十进制数、十六进制数对照表

二进制数	十进制数	十六进制数
0	0	0
1	1	1
10	2	2
11	3	3
100	4	4
101	5	5
110	6	6
111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F
10000	16	10
10001	17	11
⋮	⋮	⋮

有一点是值得一提的：人类只创造了一种数目的观念，但相同的一个数，却有好机种表示法。比如中文中的“你”这个字，英语是以“you”来表示；德语是以“du”来表示；法语中是以“vous”来表示；日语则是以“あなた”来表示。同理，“15”这个数，以二进制来表示是“1111”，以十进制来写是“15”，以十六进制来表达则是“F”，表 1.3 给出了三者之间的对照关系。

从表 1.3 中，我们可能会产生一个疑问：到底某数（如 11）是代表哪种数字系统？对于这个问题的解决方法是：我们通常会在该数字的尾部加一个英文字母（大写或小写）以示区别：

□在数尾加上 b，则表示是二进制数（取 binary 的首字母）。

□在数尾不加任何字母，则表示是十进制数。

□在数尾加上 h，则该数是十六进制数（取 hexadecimal 的首字母）。

例如：二进制数字系统的 1011b，相当于十进制数字的 11，同时也等于十六进制数字的 0Bh。

当然，如果在很容易辨别为何种数字系统的场合，数字后面的“b”或“h”是可以省略的。

1.3　二进制数、十进制数、十六进制数之间的转换关系

介绍完上节所提到的三种数字系统之后，在我们进入本小节之前，要先介绍一个“加权指数”的概念：见图 1.1。

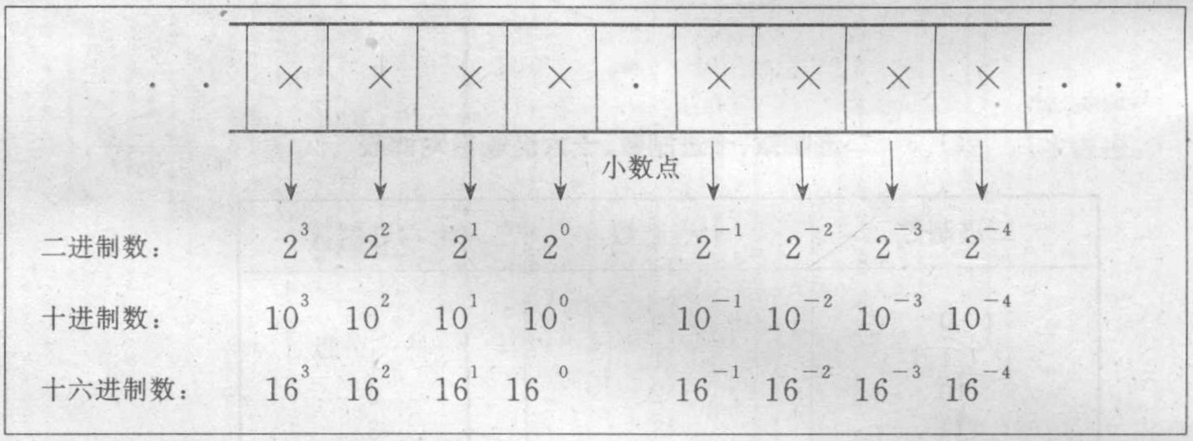


图1.1　三种不同数字系统下的加权指数大小

不同数字系统下的各个位数加权指数大小并不相同，但其原理都是一样的。以我们常用的十进制数字系统来说，个位数（第 0 位数）的加权指数就是 10 的 0 次方，而十位数就是 10 的 1 次方……第 N 位数就是 10 的 N 次方，依此类推，二进制的第 0 位数的加权指数则为 2 的 0 次方，第一位数则为 2 的 1 次方…。

实际上一般的数字都可以用加权指数的方式来书写，例如：

第3位　　第2位　　第1位　　第0位

$$1024 = 1 \times 10^3 + 0 \times 10^2 + 2 \times 10^1 + 4 \times 10^0$$

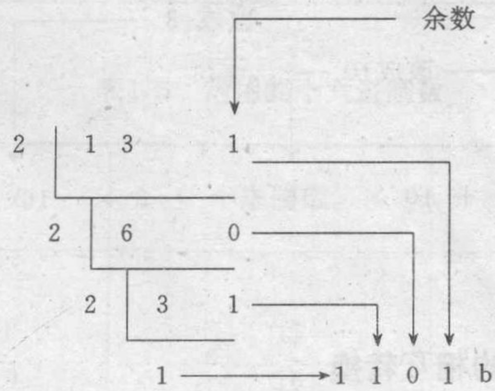
$$1010b = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$

$$A1D1h = 10 \times 16^3 + 1 \times 16^2 + D \times 16^1 + 1 \times 16^0$$

接下来，我们就可以开始进行本节的内容：各种数字系统之间的转换。

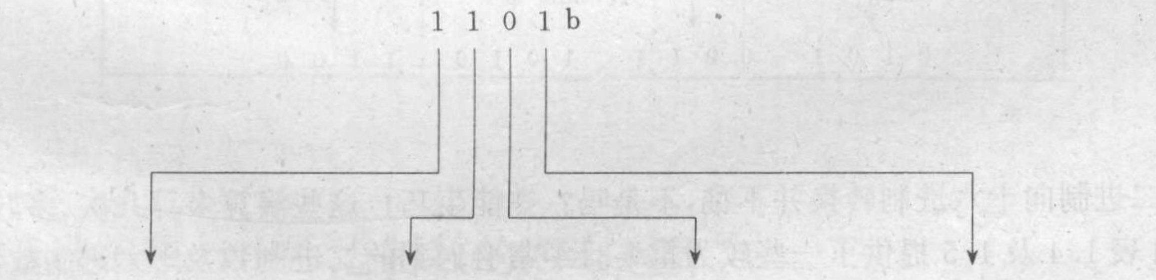
1.3.1　二进制数与十进制数之间的相互转换

例如：将 13 转换成二进制数的方法是



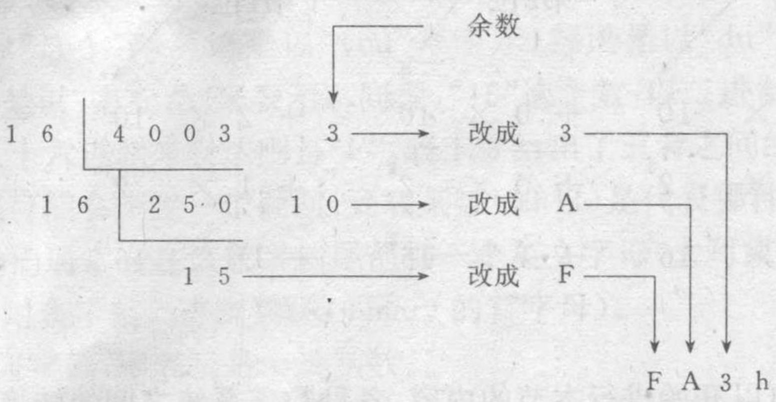
所以 13＝1101b。

同理，我们可以利用如下方法将二进制的数转换成十进制数：

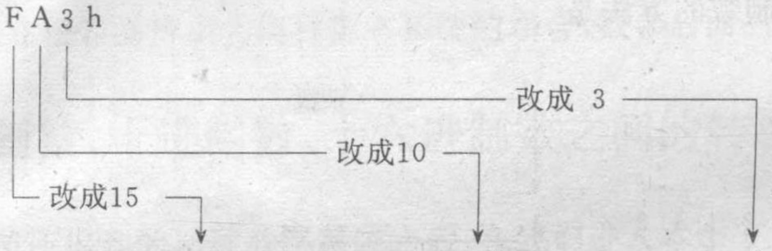


1.3.2　十六进制数与十进制数之间的相互转换

与上同理，十六进制数与十进制数之间的转换也是有规律可循的。例如：将 4003 转换成十六进制可用如下方法：



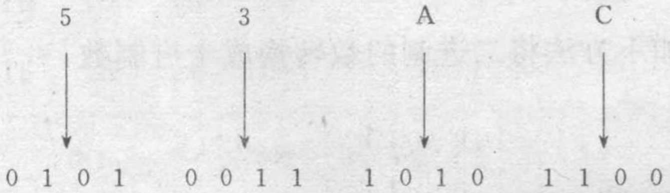
同样，以下是十六进制数转换成十进制数的方法：



$$15\times 16^{2}+10\times 16^{1}+3\times 16^{0}=4003$$

1.3.3 二进制与十六进制之间的相互转换

二进制数与十六进制数相互转换时，由最右边开始将二进制数以四个为一组，如果不满四个则依次向左补 0。例如：



可见二进制向十六进制转换并不难，不是吗？熟能生巧！这些演算多算几次、多转换几次就熟悉了！表 1.4 及 1.5 提供了一些较为重要且经常会遇到的二进制数及十六进制数，请仔细的观察！

表1.4　常用的二进制数

二进制数	十进制数	2 的乘幂
1111	15	2^4-1
10000	16	2^4
111111	63	2^6-1
1000000	64	2^6
11111111	255	2^8-1
100000000	256	2^8
1111111111	1023	$2^{10}-1$

表1.5　常用的十六进制数

十六进制数	十进制数	16 的乘幂
F	15	16^1-1
10	16	16^1
FF	255	16^2-1
100	256	16^2
7FFF	32767	$8\times16^3-1$
8000	32768	8×16^3
FFFF	65535	16^4-1
10000	65536	16^4

1.4　二进制数与十六进制数的四则运算

1.4.1　二进制数的运算

其实二进制的加减法并不难掌握，如果读者已经了解了上述的转换表，则应该已经找出了一些规律性，例如二进制数的加法如下：

```
  ↓↓  ←———— 发生进位
  1 1 0 1
+ 1 1 1 0
—————————
1 1 0 1 1
```

请注意进位发生的时机(只有两个 1 相加才进位),及计算是由右往左算(即由低位数往高位数运算)。二进制数的减法如下:

```
  ↓↓  ←———— 发生借位，向上一位
            借过来的 1，移到下
            一位时变成 2，然后
            再参与相减运算 。
  1 1 0 1
- 0 1 1 0
—————————
  0 1 1 1
```

至于乘法则是这么算:

```
        1 1 0 1
      × 0 1 1 0
      —————————
        0 0 0 0 ··········任何数与 0 相乘则为 0
      1 1 0 1   ··········任何数与 1 相乘则不变
    1 1 0 1     ··········任何数与 1 相乘则不变
+ 0 0 0 0       ··········任何数与 0 相乘则为 0
———————————————
  1 0 0 1 1 1 0 ··········最后加起来
```

除法则是这么算:

```
          1 0 ··················· 商
       ┌──────────
1 1 0 /  1 1 0 1
         1 1 0  ··················· 2 进制数相减
       ──────────
           0 0 1
           0 0 0 ··················· 2 进制数相减
       ──────────
               1 ··················· 余数
```

都很简单吧？有兴趣的话，可自行练习本章后面的习题。习而熟之，也许你会喜欢这些算法。

1.4.2 十六进制数的运算

十六进制的数也可以加、减、乘、除。但十六进制数的乘、除法直接运算起来并不方便，通常是先转换成十进制数，算好之后再转换回来。以下仅就加法与乘法来作说明，至于减法和除法读者不妨自行练习。

□加法：

```
   0 D h
 + C 9 h
 ─────────  （直接运算，如果不太习惯的话，也可以
   D 6 h     先换成 10 进制数运算，算完再换回来）
```

□乘法：

```
D h   ×   1 9 h  =   1 4 5 h
 │          │           ↑
 │ 换成十进 │           │  再将 10 进制数换回 16 进制数
 │ 制数运算 │           │
 ↓          ↓           │
1 3   ×    2 5   =    3 2 5
```

人类比较习惯十进制数的加减乘除，但电脑比较习惯二进制数的加减乘除，而十六进制数则是帮助人们去了解二进制数的一种数字。所以学习汇编语言的人要将二进制及十六进制的四则运算弄懂才行。

1.5 正负数的表示法

我们知道电脑内部充满着诸如“010101001”的信号,可以用二进制数字来表示。但前面我们所介绍的情况都是正整数的情形,如果是负数怎么办?而电脑内部并没有“+”“-”等符号可以代表正负数。为了解决这个问题,有人提出以二进制数的最高位(Bit)来表示正负号,也就是说最高位为 1 时表示是负数,为 0 时则表示正数,如表 1.6。

然而这种方法有个缺点,就是产生了+0 和-0 两个重复的数,并且原来 00000000~11111111 总共可表示 256 个数,现在只能表示 255 个数(因为+0 和-0 表示同一个数)。

表1.6 符号大小值表示法(Sign magnitude)

正数		负数	
00000000	+0	10000000	−0
00000001	+1	10000001	−1
00000010	+2	10000010	−2
0.......	.	1.......	.
0.......	.	1.......	.
0.......	.	1.......	.
01111111	+127	11111111	−127

所以又有人提出另一个方法,即所谓二进制补码——一个数的负数(补码)即是将该数的每一位取补数(即反相,将 1 变成 0,0 变成 1)再加 1。例如:

表1.7 二进制数的补码(2' s complement)**表示法**

	正数		取补数再加 1			负数(补码)	
0:	00000000	⟶	11111111	+1	⟶	00000000	:−0 (去掉进位)
1:	00000001	⟶	11111110	+1	⟶	11111111	:−1
2:	00000010	⟶	11111101	+1	⟶	11111110	:−2
3:	. . . .						
.:	. . . .						
.:	. . . .						
127:	01111111	⟶	10000000	+1	⟶	10000001	:−127
						10000000	:−128

这样就不会有+0 和-0 的问题了。并且如果把 0 放在正数这边,结果正数由 0~127,负数

由-1～-128 各有 128 个，总共 256 个。

其实这种 2 的补码的表示法可以这样看更容易理解：就是一切由 0 开始，1、2、3…就是 0＋1、1＋1、2＋1…，而-1、-2、-3…就是 0-1、-1-1、-2-1…

表 1.8　有符号数以不同数制表示的对照表

	二进制	十进制	十六进制
(126＋1)	01111111	127	7Fh
.	0.......	.	.
.	0.......	.	.
(1＋1)	00000010	2	02h
(0＋1)	00000001	1	01h
由此开始　→	00000000	0	00h
(0－1，去掉借位)	11111111	－1	FFh
(－1－1)	11111110	－2	FEh
(－2－1)	11111101	－3	FDh
.	1.......	.	.
.	1.......	.	.
(－126－1)	10000001	－127	81h
(－127－1)	10000000	－128	80h

通常正整数的表示法又称为**无符号数**(Unsigned)**表示法**，而正负数的表示法则称为**有符号数**(Signed)**表示法**(因为有＋、-号)。

一个负数前面可以无限制地加上 1，就如同一个正数前面可以任意加上 0 一样。例如处理 8 位(bit)数据时，-3 为 11111101b(或 FDh)。而处理 16 位数据时，-3 就要写成 1111111111111101b(或 FFFDh)。所以 11111101b 和 1111111111111101b 在用有符号数表示法时都代表-3，也就是说负数开头有几个 1 都可以任意加上去(二进制)。

1.6　电脑数据的基本单位：位(Bit)、字节(Byte)、字(Word)

位(Bit)是指二进制的一个位数，也就是指 0 或 1。但由于位的单位太小，不便于处理数据，因此在电脑内一般把 8 个位合成一组称为一个字节(Byte)，而以字节为单位来处理数据。在一长串的二进制数中，我们将最左边的位称为最高有效位(MSB 即 Most Significant Bit 的缩写)；而将最右边的位称为最低有效位(LSB 即 Least Significant Bit 的缩写)。

在电脑世界中，我们习惯于从右至左，由 0 开始计数，所以一个字节中的 8 个 Bits 是由 Bit0 开始向左计数直到 Bit7(这是源于英国哲学家罗素之后所产生的影响)。见图 1.2。

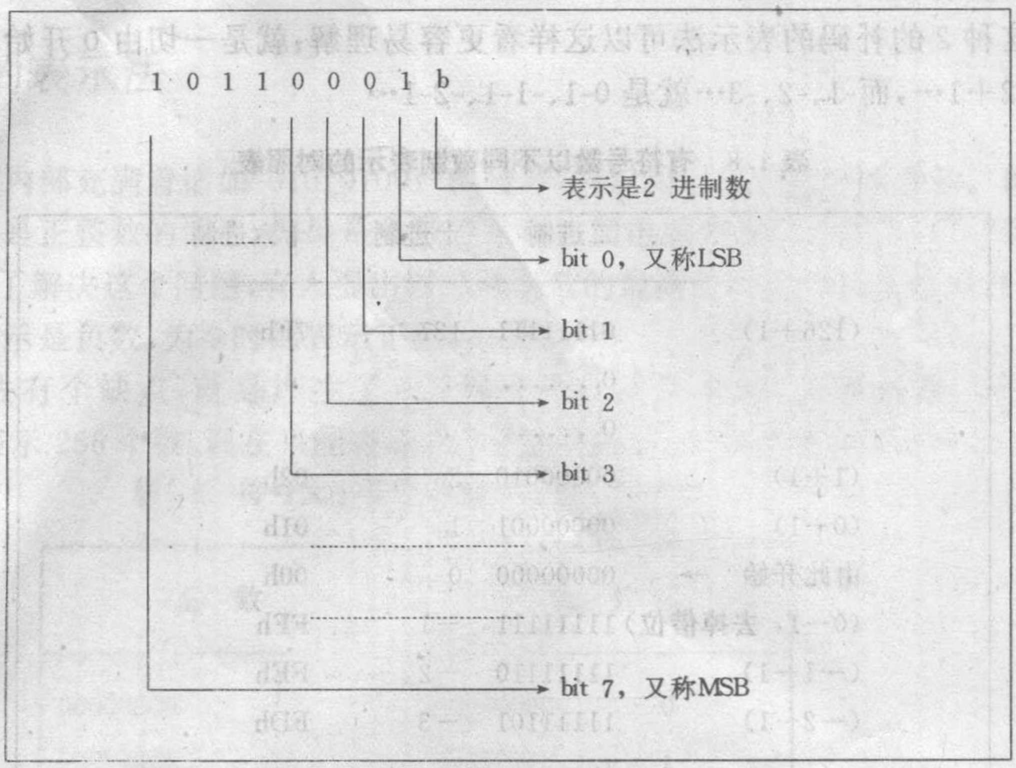


图1.2 数据的基本单位：一个字节

另外，我们一般又将两个字节合称为一个字(Word)；将两个字合称为一个双字(Double-Word，简写为DWord)；将两个双字合称为一个四字(QuadWord，简写为QWord)。请参考下表。

表1.9

名 称	bit 数
byte（字节）	8位
word（2 个字节）	16位
doubleword（4 个字节）	32位
quadword（8 个字节）	64位

此外，我们将1024个字节称为1KB(Kilo Bytes，千)，64KB称为一个段(Segment)，1024KB称为1MB(Mega Bytes，百万)，而1024MB则称为1GB(Giga Bytes，十亿)。表1.10是对上述称谓的整理。

表1.10

名　　称	字节(byte)数
word (字)	2 = 2h = 10b
dword (双字)	4 = 4h = 100b
quadword (四字)	8 = 8h = 1000b
k (Kilo 千)	1024 = 400h = 10000000000b≈1000
segment (段)	65536 = 10000h = 10000000000000000b
M (Mega 百万)	1048576 = 1000000h = 2^{20} = 100000000000000000000b≈1,000,000
G (Giga 十亿)	1073741824 = 1000000000h = 2^{30} = 1000000000000000000000000000000b ≈1,000,000,000

1.7 两种常见的符号系统:ASCII码和汉字编码

在学习了以上各种数字系统以及数据的基本单位之后,有一点我们必须了解:电脑虽然都是二进制系统,然而各个厂家生产的电脑因其内部所采用的编码系统不同而不尽相同,以致很多电脑与电脑之间无法沟通信息。有鉴于此,美国国家标准组织,制定了统一的信息交换码作为各种电脑系统之间沟通信息的标准,并规定各种电脑系统在进行数据交换之前都必须先转换成统一的 ASCII 码,如此一来,各种电脑系统就可以使用共同的语言从而达到沟通的目的。在使用中文的地区,除了 ASCII 码之外,还需要能够表示中文的汉字编码。

接下来,让我们来看一看两种常用于数据处理的符号系统:ASCII 码和汉字编码。

1.7.1 ASCII 码

ASCII 码(American Standard Code for Information Interchange 的缩写)即美国标准信息交换码。今日的个人电脑,大多采用 ASCII 码做为系统内部编码,所以大都不必再进行转换,信息交流十分方便。

通常,ASCII 码使用 7 个二进制位(Bits)来表示大小写英文字母、数字、符号等以及系统内部联系用的控制码,共 128 个(2^7=128)字符。所以,只需一个字节(Byte)就可以容得下一个 ASCII 码。

1.7.2 汉字编码

要使用电脑处理汉字,就必须将汉字转换为电脑系统能够处理的二进制编码。由于汉字太多,所以用一个字节无法表示所有的汉字,必须两个字节才够用(两字节有 256×256 种组合,

最多可以表示 65536 个汉字)。以目前常用的中文 BIG-5 码来说,为了解决与西文软件兼容的问题,因此采用了如下表所示的编码方式,能够表示 19782 个汉字(126×157)。

表 1.11 BIG-5 码编码范围

第一字节(高字节):81h—FEh (共 126)
第二字节(低字节):40h—7Eh (共 157) A1h—FEh

当汉字操作系统接收到第一个字节时,如果发现其数值在 81h~FEh 的范围内,则表示此为中文 BIG-5 码,并且接下来的字节应为该汉字编码的第二字节;如果发现第一个字节的值在 0h~7Fh 之间,则表示此字节为英文 ASCII 码。

1.7.3 概念澄清

最后,我们要不厌其烦的再次强调,电脑系统内实质上只有 0、1 两种数字,所有的数据都是 0 和 1 的组合而已,至于它们代表什么含义(什么码)?那就要看"人的解释"了。

例如:我们看到像 01000001b 这样一个 8 位数据,我们可以将它解释为一个单纯的数字,即等于十进制的 65;但是同样这么一个数据,我们也可以将它解释成 ASCII 码中的大写字母"A"。有了上述的概念之后,我们就不会被电脑中的各种表示法所迷惑了。

本章习题

1. 请解释下列名词:

二进制 (Binary System)
十进制 (Decimal System)
十六进制 (Hexadecimal System)
无符号数 (Unsigned Number)
符号位 (Sign Bit)
有符号数 (Signed Number)
二进制补码 (2's Complement)
位 (Bit)
字节 (Byte)
字 (Word)
双字 (DoubleWord)
四字 (QuadWord)
段 (Segment)
页 (Page)
节 (Paragraph)
美国标准信息交换码 (ASCII)
K 字节 (K byte, Kilobyte)
M 字节 (M byte, Megabyte, MB)
最高有效位 (MSB)

最低有效位（LSB）

2. 请将下列二进制数转换成十进制数：

（1）1100b　（2）01010101b　（3）110011100001b　（4）0001001001001000b

3. 请将下列十进制数转换成二进制数：

（1）100　（2）255　（3）1024　（4）65535

4. 请将下列十六进制数转换成十进制数：

（1）03h　（2）1EFh　（3）2DEFh

5. 请将下列十进制数转换成十六进制数：

（1）1000　（2）1024　（3）4095　（4）65535

6. 请将下列二进制数转换成十六进制数：

（1）10100111b　（2）00111100b　（3）101101011011b

7. 请做下列运算：

（1）1011b×1010b　（2）4Ch×5Ah　（3）1010b/11b

（4）5Fh-6Ch(5)　（5）5Fh＋6Ch

8. 请叙述您对有符号数、无符号数的认识。

9. 请以无符号数的方式，将下列十六进制数转换成十进制数：

（1）0Fh　（2）FDh　（3）100h　（4）50h

10. 请以有符号数方式，将下列十六进制数转换成十进制数：

（1）0Ah　（2）FFh　（3）FFFFh　（4）0F00h

11. 请计算下列二进制数的补码：

（1）11100001b　（2）10111100b

12. 请查出下列字符的 ASCII 码：

A、B、C、a、b、c、1、2、3、*、&、%、BELL、LF、CR

13. 请查出下列 ASCII 码所代表的字符：

41h、65h、32h、33h、34h、4Ch、07h、08h、0Ah、0Dh

第二章 PC的基本构造

要学好汇编语言，除了需要了解数字符号系统外，也必须熟悉CPU及存储器的结构。本章即针对此目的，将PC的基本结构做一扼要的介绍，为读者打好学习汇编语言的基础。本章的主要内容有：

□PC的基本构造

□80x86 CPU及其寄存器组

□存储器的结构

□80x86的分段式存储器管理

□80x86指令的寻址方式

个人电脑(PC)基本上是由存储器、中央处理器以及输入/输出设备等部分所组成，其中以中央处理器为整个电脑的核心。现在我们就从PC的基本构造开始了解。

2.1 PC的结构简介

基本上，PC是由存储器(Memory)、中央处理器(CPU)、输入/输出设备(Input/Output)等部分组成。其中，输入设备负责由外界输入数据到电脑内部，CPU执行程序以处理数据，存储器则用来存放程序及运行程序时所需的数据，当数据处理完毕，则将结果由输出设备输出。如图2.1。

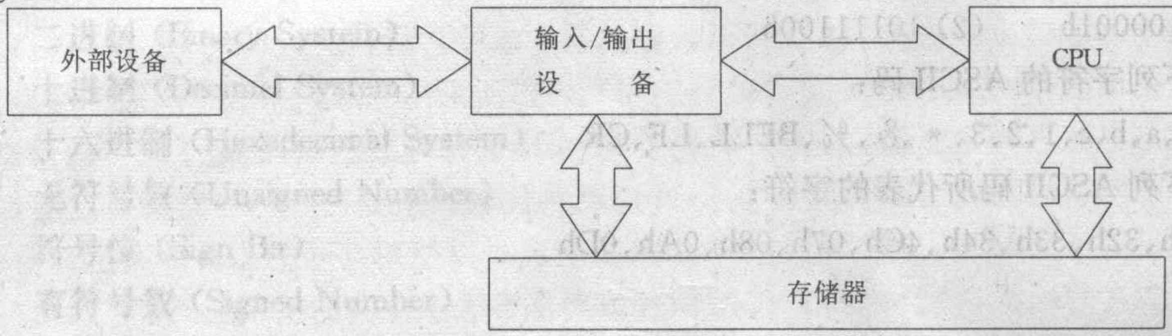


图2.1　PC的基本结构

2.2 80x86 CPU及其寄存器组

2.2.1 80x86 CPU

CPU是电脑的核心，它能快速地执行程序指令，从而使电脑发挥出高超的性能。80x86 CPU是由Intel公司开发的系列微处理器，包括8088/86/286/386/486/Pentium(俗称586)等，简称为80x86系列。80x86 CPU系列具有下列特性：

□互相兼容的指令集；

□互相兼容的寄存器结构；

□互相兼容的存储器管理方式。

所以 8088/86/286/386/486/Pentium 这些 CPU，依序具有向上(upward)兼容性。凡是为 8088 所编写的程序，均可以在 8086/286/386/486/Pentium 系统上执行；为 8086 所编写的程序可以在 80286/386/486/Pentium 上执行，依此类推(反之则不具备兼容性)。也就是说，高档的 CPU 可以接受较低档 CPU 的指令，而较低档的 CPU 却无法执行高档 CPU 的某些指令。

2.2.2 80x86 的寄存器组

CPU 是一个负责执行指令的单元，为了方便执行指令起见，CPU 内部也设置了一些存储部件，用于临时性地储存一些数据，这种 CPU 内部的存储部件称为寄存器(Register)，见图 2.2。通常，对于程序员来讲，只要学习了 CPU 的指令，并且熟悉各个寄存器的用法，便可以顺利地控制 CPU 的动作。

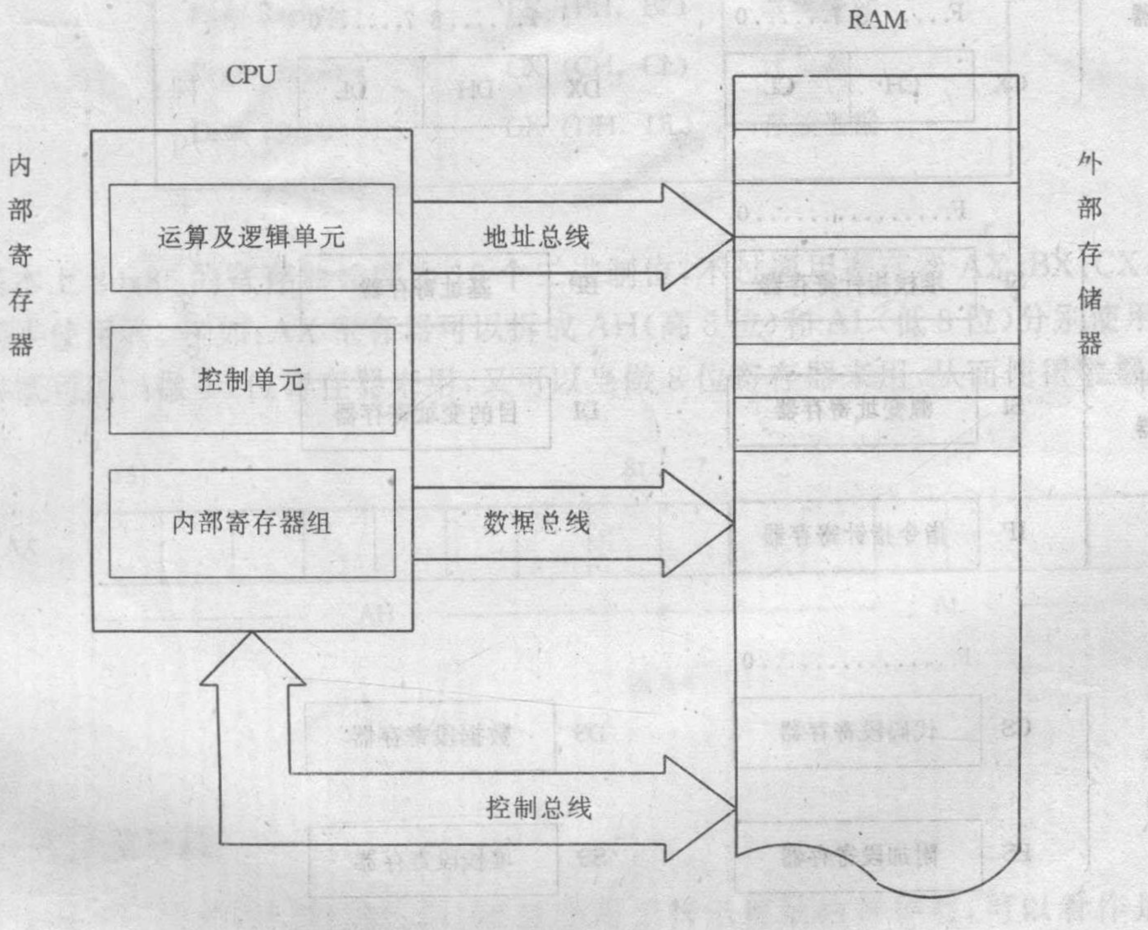


图2.2 寄存器是 CPU 内部的储存单元，不同于外部的存储器

在 80x86 的实模式(Real Mode，即 8086 的执行模式)下，CPU 内部共有 14 个寄存器，可分为 4 组：

□通用寄存器：AX、BX、CX、DX

□指针寄存器：SP、BP、SI、DI、IP

□段寄存器：CS、DS、ES、SS

□标志寄存器：FLAG

说明

80286～Pentium CPU 都有两种执行模式，一种是原来 8086 的执行模式，称做实模式（Real Mode），另一种称做保护模式（Protected MOde）。本书的内容是针对实模式下的程序设计而言。

所有这些寄存器的长度都是 16 位的，但是通用寄存器可以拆成两个 8 位的寄存器 AX＝（AH，AL）、BX＝（BH，BL）、CX＝（CH，CL）、DX＝（DH，DL）来使用，见图 2.3。

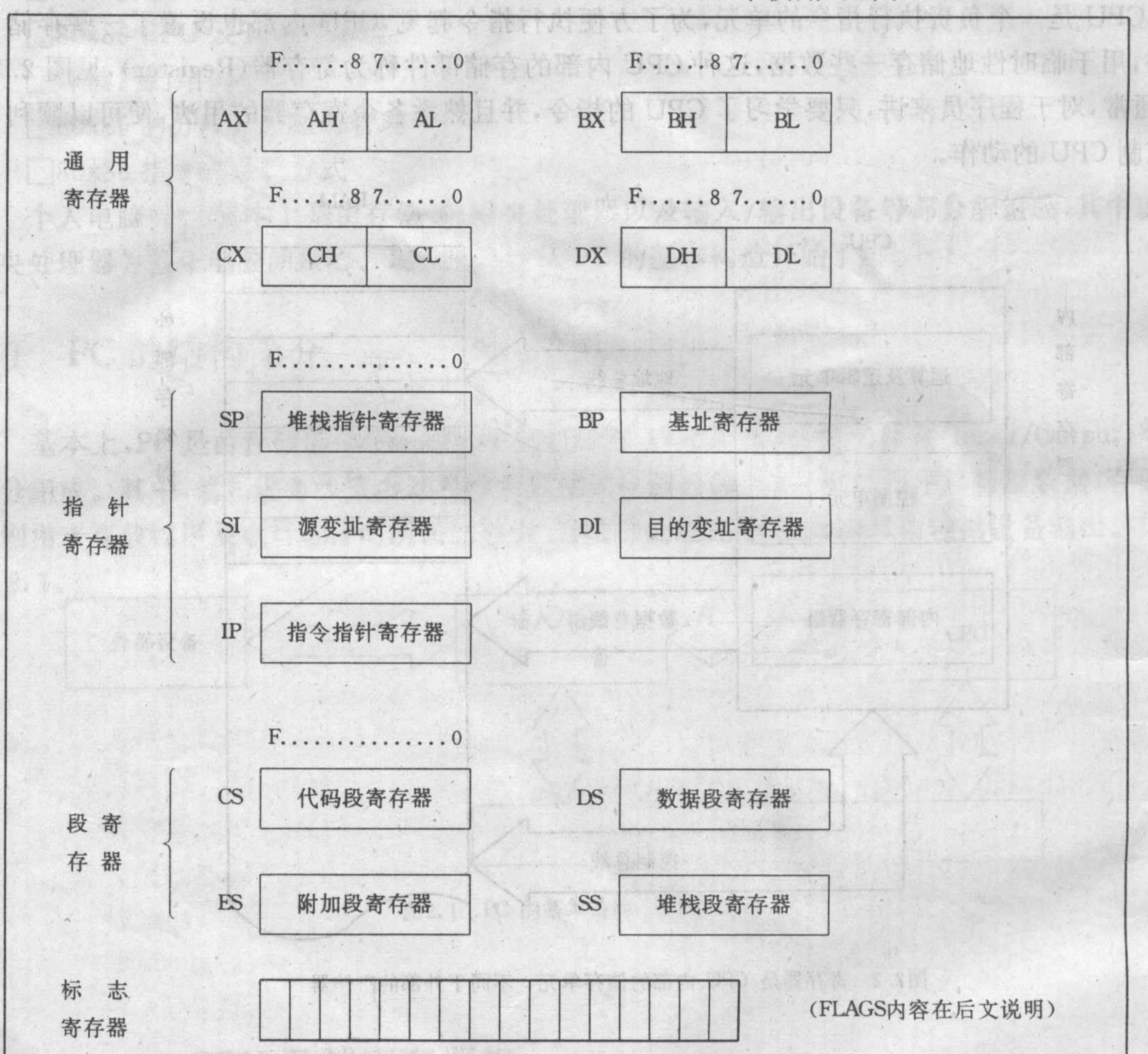


图2.3　四大类寄存器

图 2.3 的寄存器组中，如果只是做为存放数据而用，就称为**寄存器**，如果做为地址指示（存放存储器地址）用途者，则称为**指针**（Pointer），而专门用来存放段地址的，我们将其称为**段寄存器**。

2.2.3 通用寄存器

AX、BX、CX、DX 称为通用寄存器，它们通常被用来临时存放运算过程中所需的数据及运算结果。这些用途可以从它们的英文名称看出：AX 是用来存放运算结果(Accumulator)；BX 则做为基址寻址时的基地址寄存器(Base)；CX 多用于程序循环时的计数器(Counter)；DX 则做为数据(Data)寄存器。其整理见表 2.1。

表2.1 通用寄存器的用途

英文名称	简　　称	一般用途
Accumulator	AX (AH, AL)	存放运算结果
Base Register	BX (BH, BL)	基址寻址
Count Register	CX (CH, CL)	计数器
Data Register	DX (DH, DL)	存放数据

基本上 80x86 的寄存器长度为 16 个二进制位，不过通用寄存器 AX、BX、CX、DX 是可以拆成两半使用的。例如：AX 寄存器可以拆成 AH(高 8 位)和 AL(低 8 位)分别使用。由于这些寄存器既可以当做 16 位寄存器来用，又可以当做 8 位寄存器来用，从而使得数据的存取更具有弹性。

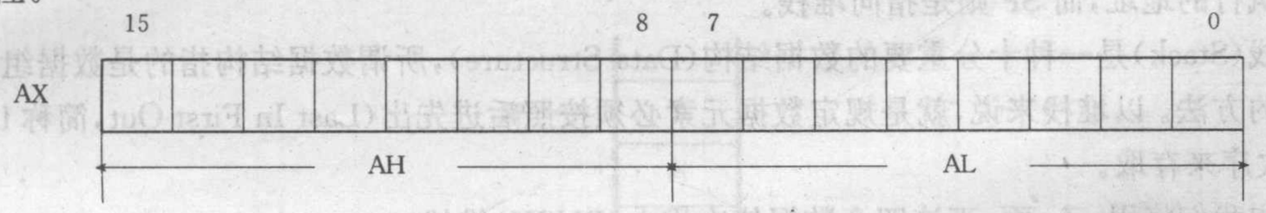


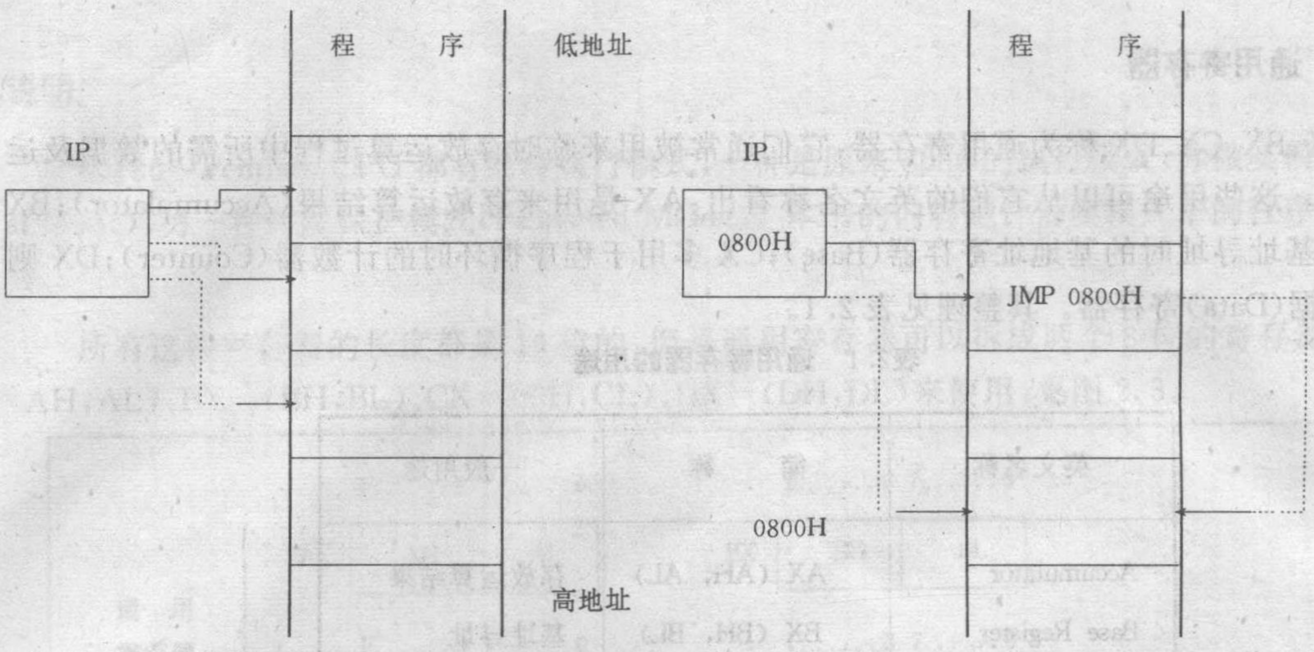
图2.4

2.2.4 指针寄存器

80x86CPU 包含 IP、SP、BP、SI、DI 五个用于指示地址的寄存器，可以看作是地址的“指针”，故称做指针寄存器，它们用于存放存储器的地址。以下为这些寄存器的用途说明。

IP

IP(Instruction Pointer)称为指令指针寄存器，负责掌管 CPU 执行程序的流程。当 CPU 执行完一个指令后，IP 便指向下一个指令的地址，接着 CPU 便会往 IP 所指的地址去读取所要执行的指令。诸如 JMP(跳转指令，使程序跳到其它地方执行)、CALL(调用子程序)等控制 CPU 走向的指令，其实便是以改变 IP 的值来控制 CPU 的执行流程。图 2.5 说明了 IP 指令指针控制器如何控制 CPU 的执行流程。



(a) 改变 IP 的指向，顺序读取指令　　(b) JMP指令改变 IP 指向，跳到指定的地址

图2.5

SP

SP(Stack Pointer)称为堆栈指针寄存器，SP 与 IP 有些类似，只不过差别在于 IP 是用来指向程序执行的地址，而 SP 则是指向堆栈。

堆栈(Stack)是一种十分重要的数据结构(Data Structure)，所谓数据结构指的是数据组织和管理的方法。以堆栈来说，就是规定数据元素必须按照后进先出(Last In First Out，简称 LIFO)的次序来存取。

比如我们将甲、乙、丙、丁这四个数据依次堆入(PUSH)堆栈：

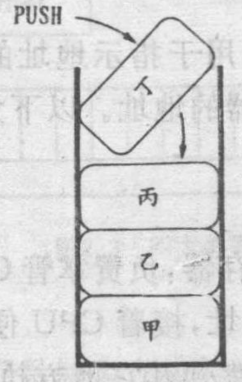


图 2.6　堆栈的 PUSH 动作

则当我们想由堆栈取出(POP)数据时，其次序则应是丁、丙、乙、甲：

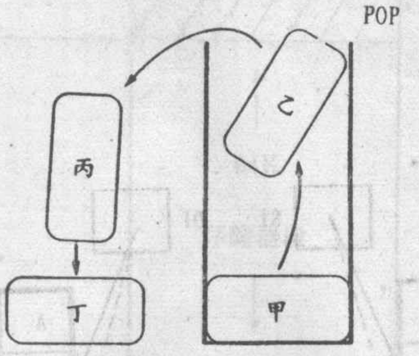


图 2.7　堆栈的 POP 动作

也就是说，像“丁”这个最后被 PUSH 的数据是最先被 POP 的。这是由于堆栈是一种单一出入口的数据结构，自然就会造成“后进先出”的特性。

堆栈是一种把“后来数据”压在“先前数据”上的数据结构，最后被 PUSH 进去的数据总是会位于堆栈的最顶端。而 SP 便是用来记录堆栈最顶端的地址的寄存器。

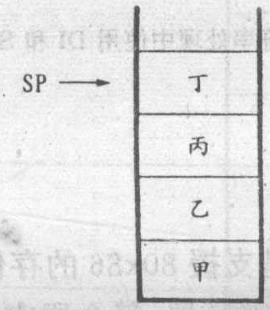


图 2.8　堆栈指针的指向

BP

BP(Base Pointer)与 SP 一样，也可以用做堆栈的指针。不过 SP 总是指向堆栈的顶端，而 BP 则可以由我们通过设定而使之指向堆栈内的任何位置，如此一来，我们就可以很方便地存取堆栈内任意位置的数据，而不再局限于 SP 所指向的栈顶数据。

DI 与 SI

DI 和 SI 是用于地址索引(Index)的两个指针寄存器，分别称源变址寄存器和目的变址寄存器。80x86 有许多寻址方法要用到 DI 和 SI 这两个指针寄存器。例如在 80x86 的字符串处理指令中，SI 用于索引源(Source)字符串的地址，而 DI 则是用于索引目的(Destination)字符串的

地址。这样,利用这两个指针寄存器我们就可以很方便的处理字符串中的每一个字符了。

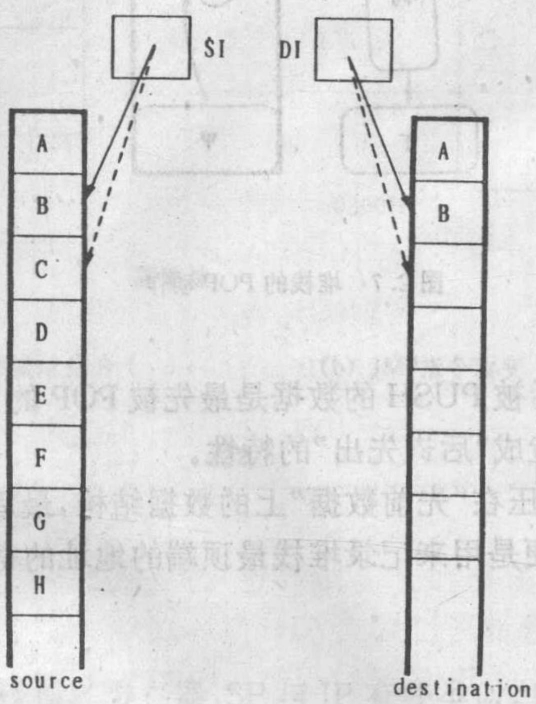


图 2.9 字符串处理中使用 DI 和 SI

2.2.5 段寄存器组

这是 80x86 独特的寄存器组,其用途是支援 80x86 的存储器管理功能。在 80x86 中,存储器是分段管理的,CS、DS、ES、SS 是四个段寄存器,每个段寄存器均定义出一个存储器段的起始地址。一个段是指在存储器中,由段寄存器所指的位置算起最高可达 64K 范围的区域。见图 2.10。

80x86 的四个段寄存器彼此之间是完全自由而独立的,各自定义的存储器段是互不影响的。也就是说,当一个段被设定之后,并不影响另一个段寄存器在相同的地址上进行段的设定。所以段与段之间可以是完全分离、相互衔接、部分重叠甚至是完全重叠的。见图 2.11。

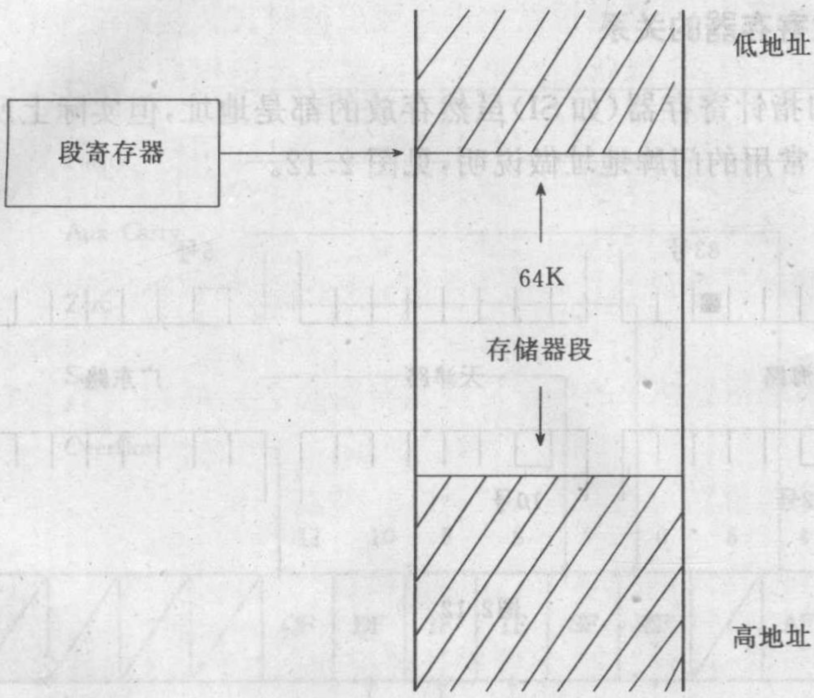


图2.10　段寄存器的范围

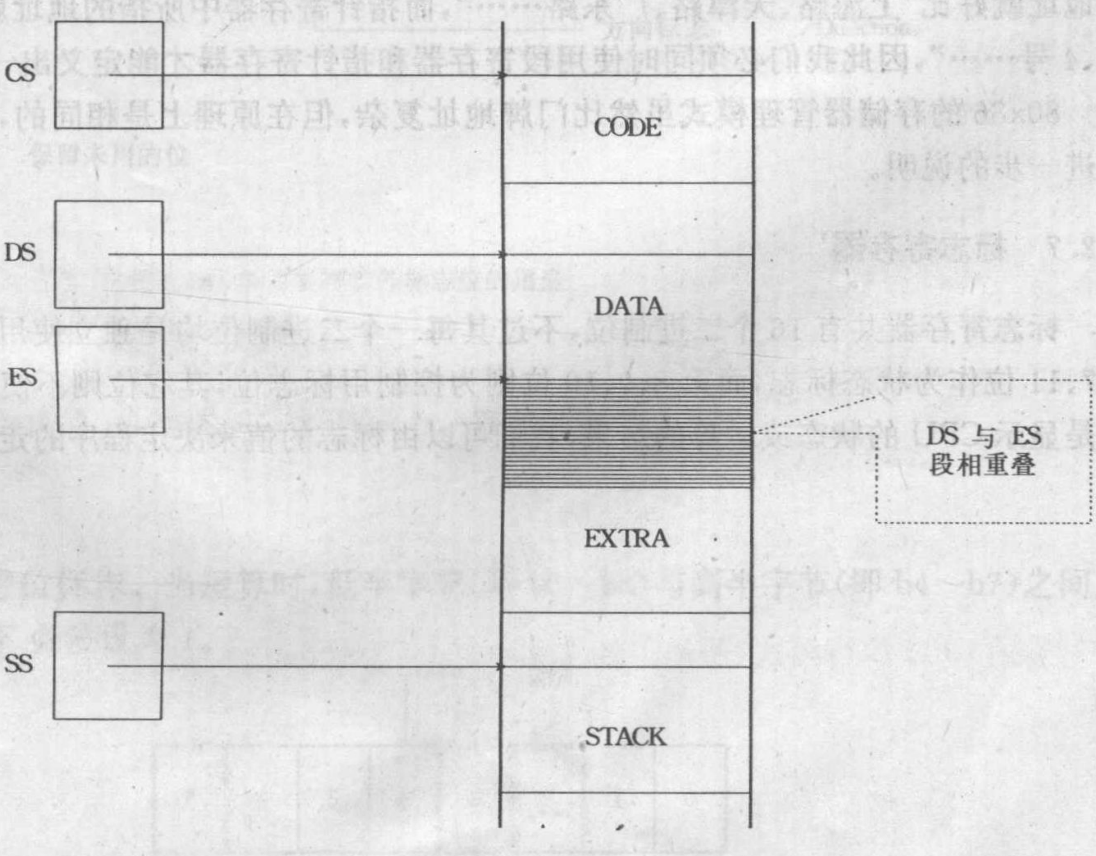


图2.11　四个段寄存器的范围可以分离或重叠

2.2.6 指针寄存器与段寄存器的关系

段寄存器(如 DS)和指针寄存器(如 SI)虽然存放的都是地址,但实际上所储存的地址在意义上并不相同,我们以常用的门牌地址做说明,见图 2.12。

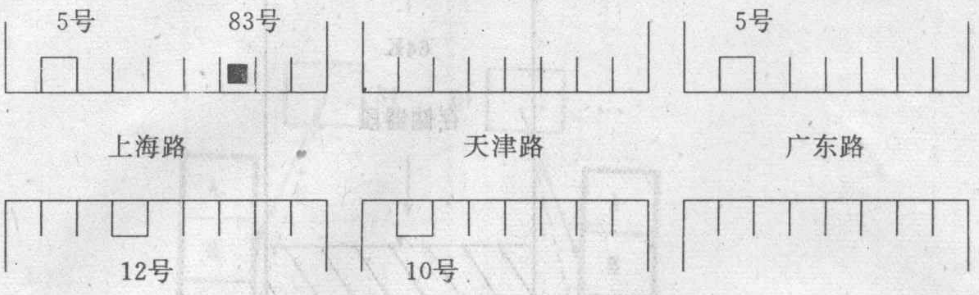


图2.12

电脑中的存储器组织方式,类似于一般住址的安排方式,如果我们将某个城市中的每一户人家都视为一个存储单元,则门牌号码便是各存储单元的地址。以上图来看,当我们想要取得有■标记的存储器单元中的数据时,我们必须正确地指出该存储器的地址是"上海路 83 号",而不能简单地把"83 号"或"上海路"当成该存储器单元的地址。由此可见,段寄存器中所储存的地址就好比"上海路、天津路、广东路……",而指针寄存器中所指的地址就如同"1 号、2 号、3 号、4 号……",因此我们必须同时使用段寄存器和指针寄存器才能定义出一个完整的地址。

80x86 的存储器管理模式虽然比门牌地址复杂,但在原理上是相同的,我们将在第三节中作进一步的说明。

2.2.7 标志寄存器

标志寄存器共有 16 个二进制位,不过其每一个二进制位均是独立使用的。其中第 0、2、4、6、7、11 位作为状态标志,而第 8、9、10 位则为控制用标志位,其它位则不使用。标志的主要作用是显示 CPU 的状态或运算的结果,我们可以由标志的值来决定程序的走向。

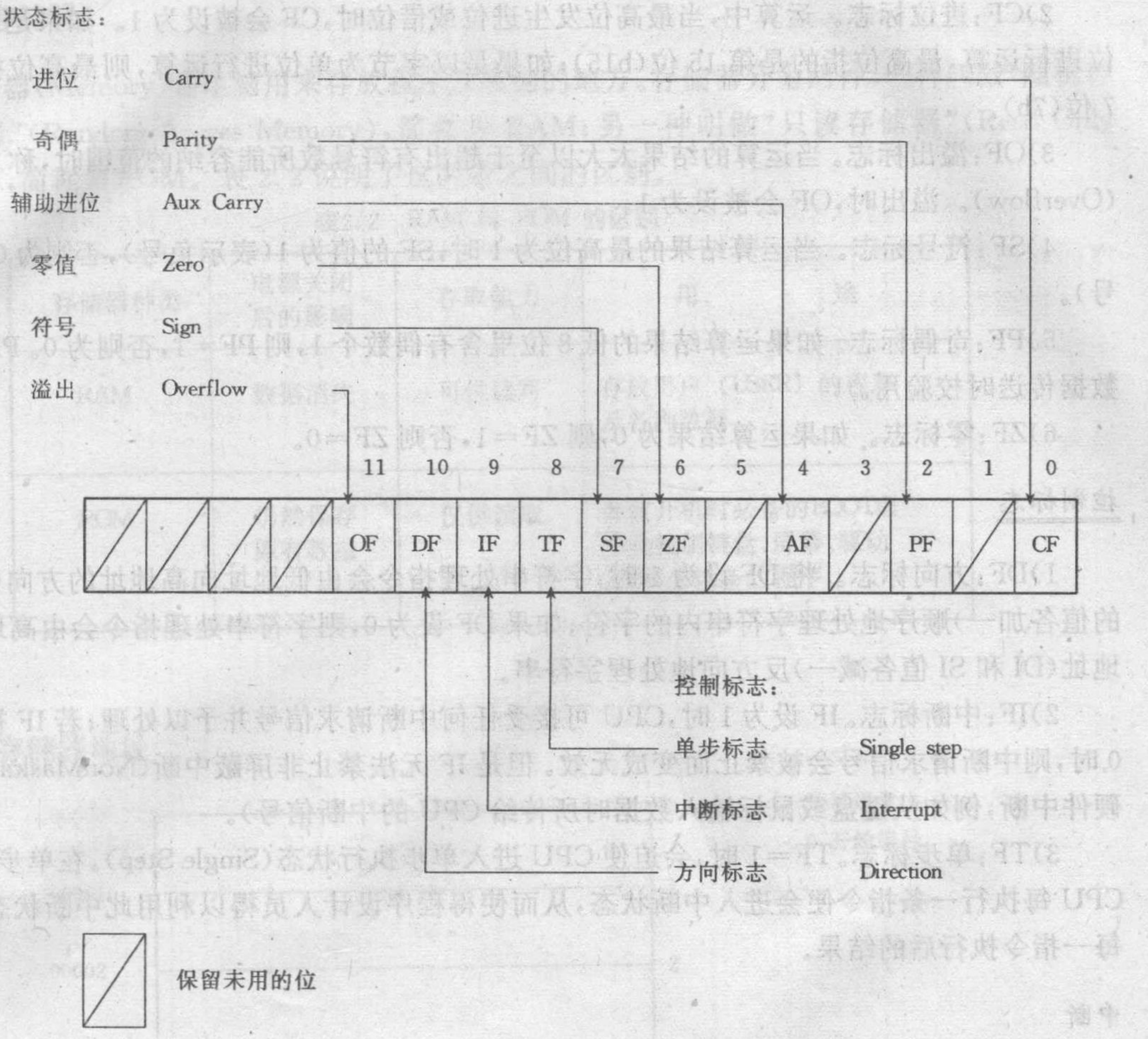


图2.13　标志寄存器各标志位的用途

标志位共分为两类：状态标志和控制标志。现分述如下：

<u>状态标志</u>

1)AF：辅助进位标志。当运算时，低半字节(即 b0～b3)与高半字节(即 b4～b7)之间发生进位或借位时，AF 会被设为 1。

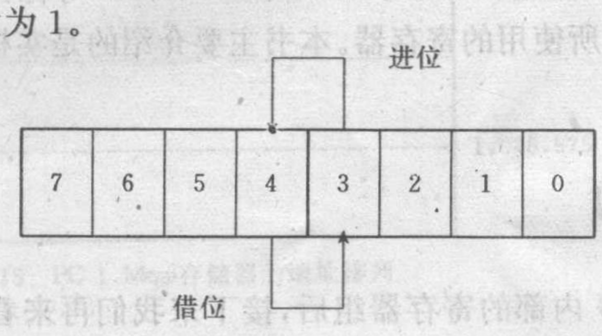


图2.14

2)CF:进位标志。运算中,当最高位发生进位或借位时,CF 会被设为 1。如果是以字为单位进行运算,最高位指的是第 15 位(b15);如果是以字节为单位进行运算,则最高位指的是第 7 位(7b)。

3)OF:溢出标志。当运算的结果太大以至于超出有符号数所能容纳的范围时,称之为溢出(Overflow)。溢出时,OF 会被设为 1。

4)SF:符号标志。当运算结果的最高位为 1 时,SF 的值为 1(表示负号),否则为 0(表示正号)。

5)PF:奇偶标志。如果运算结果的低 8 位里含有偶数个 1,则 PF=1,否则为 0。PF 可做为数据传送时校验用。

6)ZF:零标志。如果运算结果为 0,则 ZF=1,否则 ZF=0。

控制标志

1)DF:方向标志。将 DF 设为 1 时,字符串处理指令会由低地址向高地址的方向(DI 和 SI 的值各加一)顺序地处理字符串内的字符;如果 DF 设为 0,则字符串处理指令会由高地址向低地址(DI 和 SI 值各减一)反方向地处理字符串。

2)IF:中断标志。IF 设为 1 时,CPU 可接受任何中断请求信号并予以处理;若 IF 被清除为 0 时,则中断请求信号会被禁止而变成无效。但是 IF 无法禁止非屏蔽中断(NonMaskable,属于硬件中断,例如从键盘或鼠标输入数据时所传给 CPU 的中断信号)。

3)TF:单步标志。TF=1 时,会迫使 CPU 进入单步执行状态(Single Step)。在单步状态下,CPU 每执行一条指令便会进入中断状态,从而使得程序设计人员得以利用此中断状态来检查每一指令执行后的结果。

中断

80x86 具有处理中断(Interrupt 或称 Trap)的能力。当中断请求信号出现时,CPU 便会"中断"当前的工作转而去执行中断所指向的程序(这种程序叫中断服务程序)。在汇编语言中可使用 INT 指令来产生中断。

2.2.8 80386/486 的寄存器组

80386/486 是 32 位的 CPU,除了将段寄存器以外的所有寄存器均扩充为 32 位之外,并增加了两个段寄存器(GS、FS)以及 IDTR、GDTR、LDTR、TR 等保护模式下(Protected Mode,一种特殊的 CPU 执行模式)所使用的寄存器。本书主要介绍的是实模式,保护模式不在讨论的范围。

2.3 存储器的结构

在了解了 80x86 CPU 内部的寄存器组后,接下来我们再来看看在 80x86 系列下,其存储器又是怎样安排的呢?并且也了解一下 CPU 如何与外部存储器交换数据。

2.3.1 存储器

存储器(Memory)是电脑用来存放程序及数据的地方。存储器分为两种,一种叫做"随机存取存储器"(Random Access Memory),简称为 RAM;另一种叫做"只读存储器"(Read Only Memory),简称为 ROM。表 2.2 说明了这两者之间的区别。

表2.2 RAM 与 ROM 的区别

存储器种类	电源关闭后的影响	存取能力	用途
RAM	数据消失	可供读写	存放用户(USER)的程序及各种数据。
ROM	仍然保存原有数据	仅供读取	存放开机时必备的BOOT程序及控制键盘、屏幕、驱动器等I/O设备的程序。

2.3.2 存储器地址

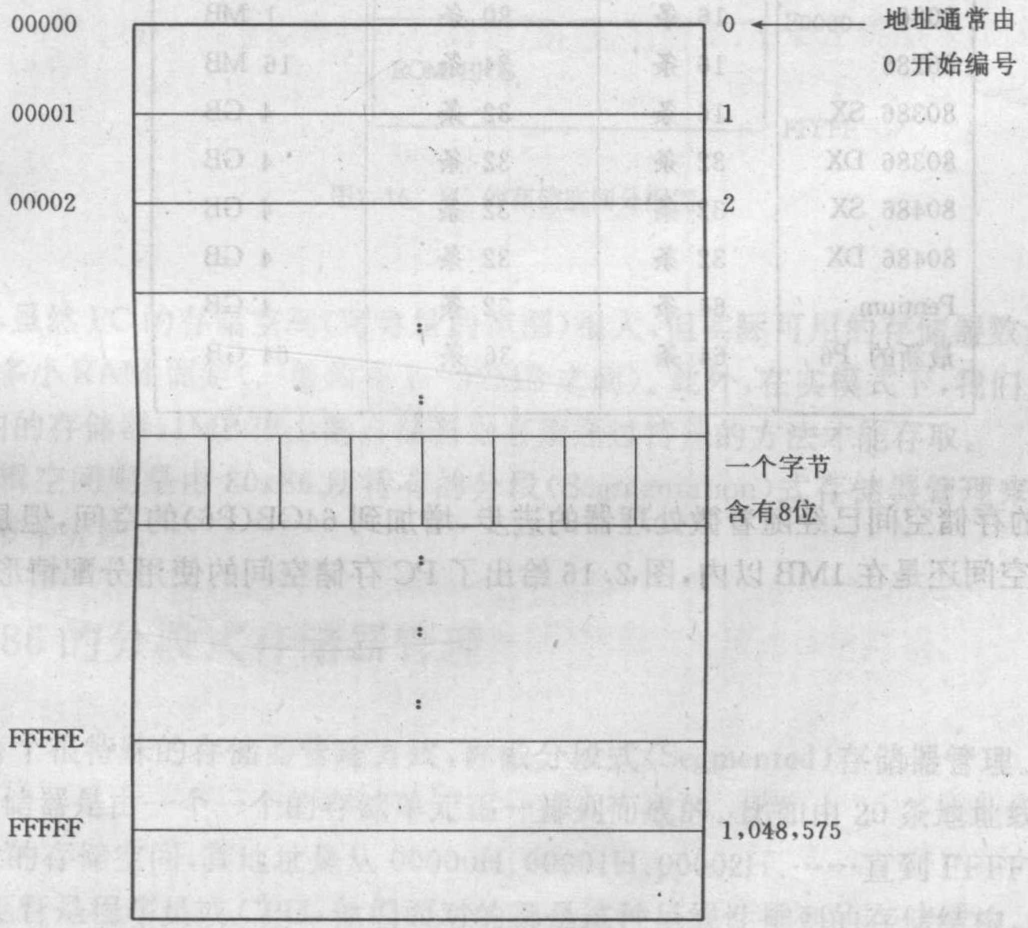


图2.15 PC 1 Mega存储器的地址排列

由于电脑的存储量很大(由一百万至数十兆字节),为了便于管理,我们通常把存储器单元排列编号。以 8088/86 1MB(1 兆字节)的存储器为例,我们把它编号成 00000H～FFFFFH,称

为存储器地址(Address),就如同我们把门牌编号叫做地址一样。

PC 存储器的地址排列,就如图 2.15 一样,是一个字节一个字节排列的,因此程序和数据也是顺序地存储在存储器中。

2.3.3 存储空间

一台电脑所能够管理的存储器范围称为存储空间(Memory Space),存储空间可分为**物理空间**和**逻辑空间**两个层次。

物理存储空间的大小与 CPU 有关,8088/86 有 20 条地址线能够寻址 $2^{20}=1M$(,百万)个字节,所以其存储空间为 1M。而 80286 有 24 条地址线,所以可寻址 $2^{24}=16M$ 的存储空间。至于 80386/486/Pentium 则有 32 条地址线,所以其存储空间的大小为 $2^{32}=4Giga$(十亿)。请参考表 2.3。

表2.3 80x86 系列的寻址空间

CPU	数据线 (Data Bus)	地址线 (Address Bus)	寻址能力
8088	8 条	20 条	1 MB
8086	16 条	20 条	1 MB
80286	16 条	24 条	16 MB
80386 SX	16 条	32 条	4 GB
80386 DX	32 条	32 条	4 GB
80486 SX	32 条	32 条	4 GB
80486 DX	32 条	32 条	4 GB
Pentium	64 条	32 条	4 GB
最新的 P6	64 条	36 条	64 GB

虽然 PC 的存储空间已经随着微处理器的进步,增加到 64GB(P6)的空间,但是在 DOS 下最常用的存储空间还是在 1MB 以内,图 2.16 给出了 PC 存储空间的使用分配情形。

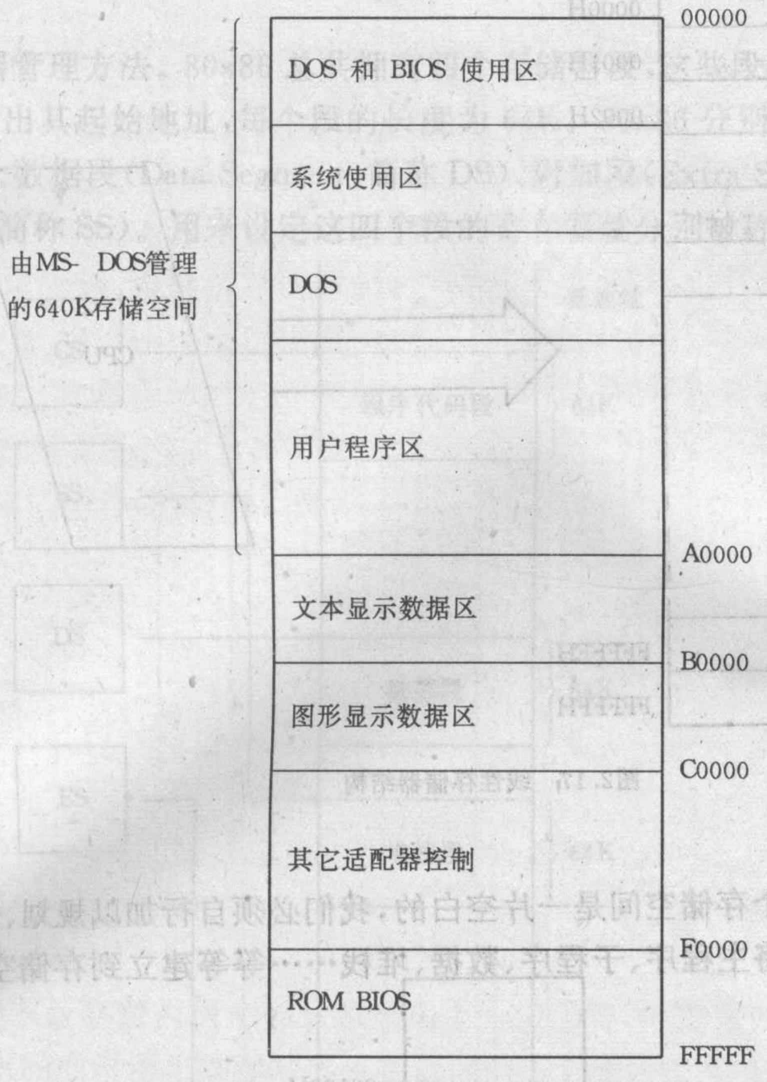


图2.16 PC 的存储空间分配图

请注意，虽然 PC 的存储空间(可寻址的范围)很大，但实际可用的存储器数量却必须视主机板上插有多小 RAM 而定(一般约在 1～32MB 之间)。此外，在实模式下，我们只能直接存取到 1MB 以内的存储器，1MB 以上的存储器则必须通过特殊的方法才能存取。

至于逻辑空间则是由 80x86 所特有的分段(Segmentation)式存储器管理来负责处理，我们将在下一节中介绍。

2.4 80x86 的分段式存储器管理

80x86 有个很特殊的存储器管理方式，称做分段式(Segmented)存储器管理。在以往的电脑结构中，存储器是由一个一个的存储单元逐一排列而成的。比如由 20 条地址线所建立的 $2^{20}=1M$ 个地址的存储空间，其地址是从 00000H、00001H、00002H、……直到 FFFFFH 呈一直线的排列着。不管是程序员或 CPU，他们面对的都是这种呈线性排列的存储结构，见图 2.17。

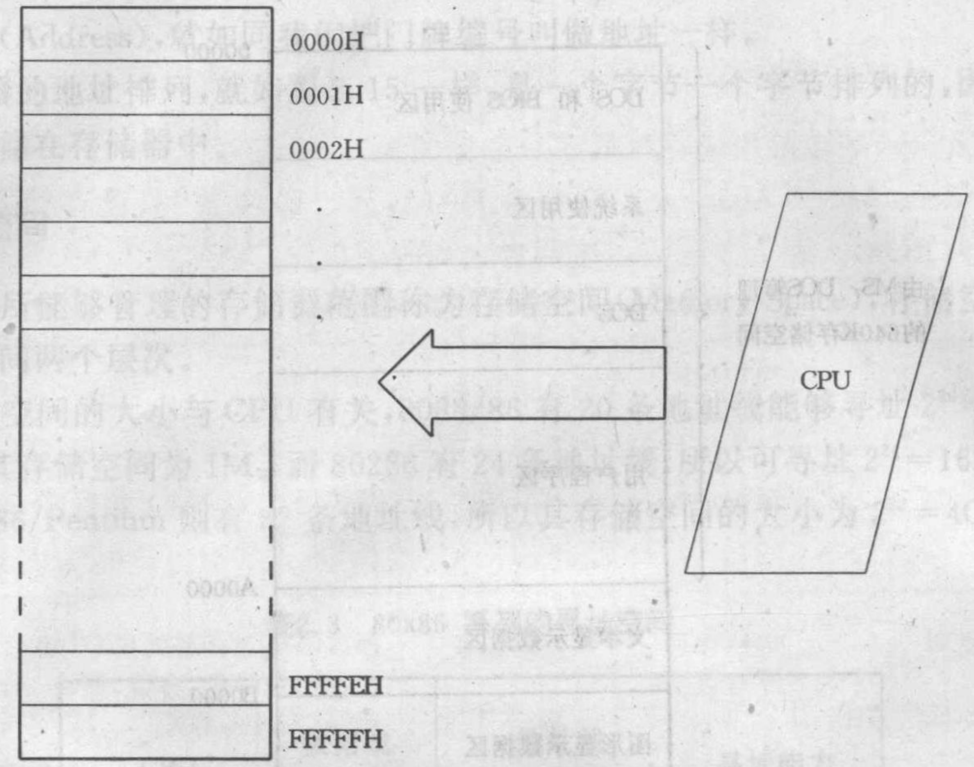


图2.17 线性存储器结构

在这种结构下，整个存储空间是一片空白的，我们必须自行加以规划、安排，以区分出不同用途的区域，然后才能将主程序、子程序、数据、堆栈……等等建立到存储空间内，见图 2.18。

图2.18 存储空间规划举例

但是这样的规划是由程序员所安排的，CPU 可一点也不知道，所以程序员必须时常注意不可破坏既定的存储器规划，时时刻刻要维持存储空间的秩序，这是程序员的职责，系统是不负任何责任的！

为了减轻程序员的负担，同时也为了强化系统的功能，于是人们便采用了分段式(Seg-

mentation)的存储器管理方法。80x86 总共拥有四个存储器段，这些段各以一个段寄存器(Segment Register)来指出其起始地址，每个段的长度为 64K。80x86 分别称为：程序代码段(Code Segment，简称 CS)、数据段(Data Segment，简称 DS)、附加段(Extra Segment，简称 ES)、堆栈段(Stack Segment，简称 SS)。用来设定这四个段的寄存器就分别被称为 CS、DS、ES、SS。

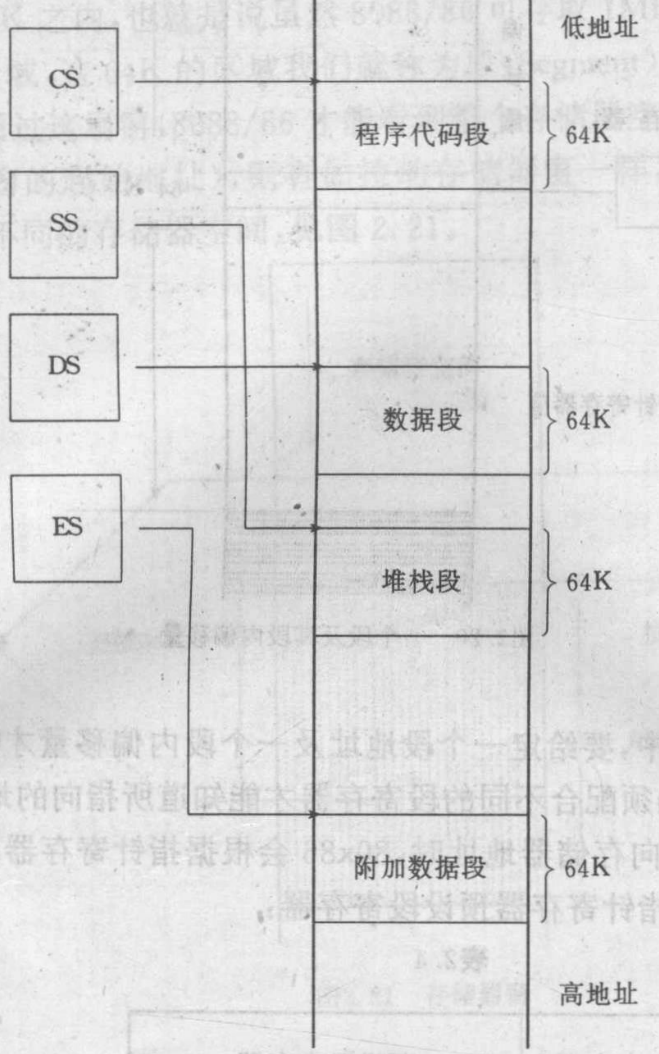


图2.19 80x86 的 4 个存储器段

平常 80x86 要从存储器中存取数据，便是通过这 4 个段寄存器来指向存储器地址的。例如：当 CPU 要读取指令时，首先要给出 CS 所指的段地址(即程序段的起始地址)，然后才从该段中读取指令。而当 CPU 要读取数据时，则从 DS 所指向的段内读取数据。

2.4.1 段地址和段内偏移量

在分段式存储器管理的系统内，一个存储器地址是由“段地址”和“段内偏移量”所构成的。段地址就是段寄存器所指向的地址，也就是一个段的起始地址；而段内偏移量则是一个段内部的地址，就是以段寄存器为参考点(起算点)来计算的，所以在英文中称为偏移(offset)地址，这个段内偏移量是依靠指针寄存器来储存的。

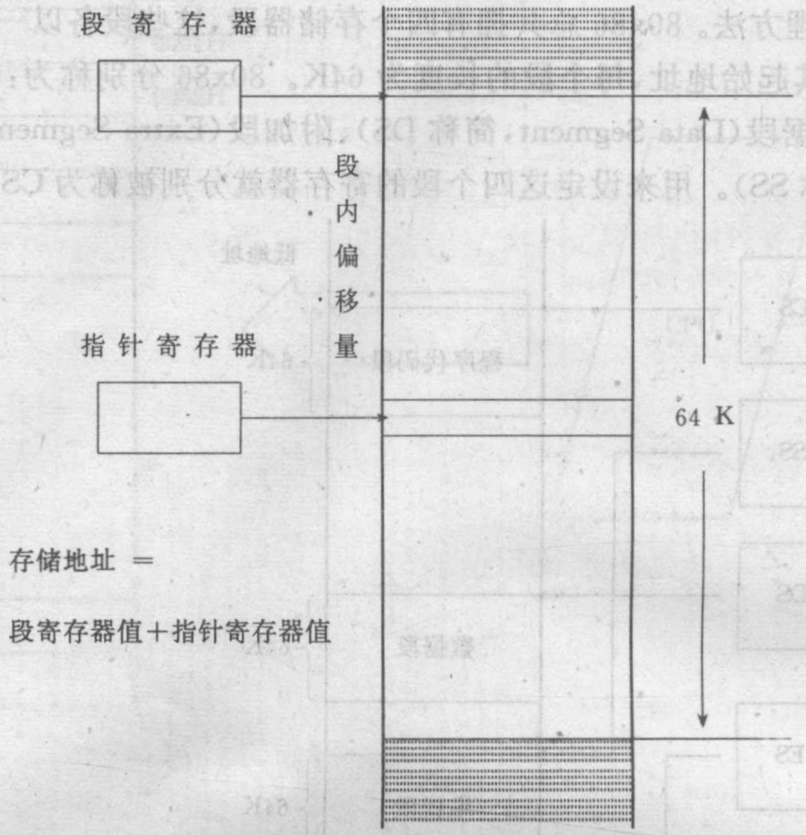


图2.20　一个段及其段内偏移量

由于在分段式管理的系统中，要给定一个段地址及一个段内偏移量才能完全定出一个地址来，因此每一个指针寄存器仍须配合不同的段寄存器才能知道所指向的地址。如果不指明段寄存器而仅以指针寄存器来指向存储器地址时，80x86 会根据指针寄存器的不同而预设一个段寄存器来配合。下表列出各指针寄存器预设段寄存器：

表2.4

指针寄存器	预设段寄存器
IP	CS
SP	SS
BP	SS
SI	DS
DI	DS或ES，视状况而定
BX	DS

举例来说，当我们指明要取得 SI 地址的数据而并未指明段寄存器时，实际上所取得的资料是位于 DS:SI 所指向的地方①。

① 在通用寄存器组中，仅 BX 可当成指针寄存器用。

2.4.2 存储器窗

上述这种组合式寻址方式，我们可以用存储器窗的方式加以说明。以 8088/86 为例，其地址是由段寄存器和段内偏移量所组成的，一旦段寄存器指向一个固定位置后，段内偏移量所能变化的范围就在 64K 之内，也就是说虽然 8088/86 可存取 1MB 的存储器空间，但一次最多只能"看"到 64K 的区域，这 64K 的区域我们就称为段(Segment)。由于一个段看起来象一扇窗(称为存储器窗)，通过这扇窗，8088/86 才能看到整个存储器空间。而且，更改段寄存器所指向的地址(即存储器窗的起始地址)，则有如拉动存储器窗一样，将存储器窗拉动到其它位置，8088/86 才能看到不同的存储器空间，见图 2.21。

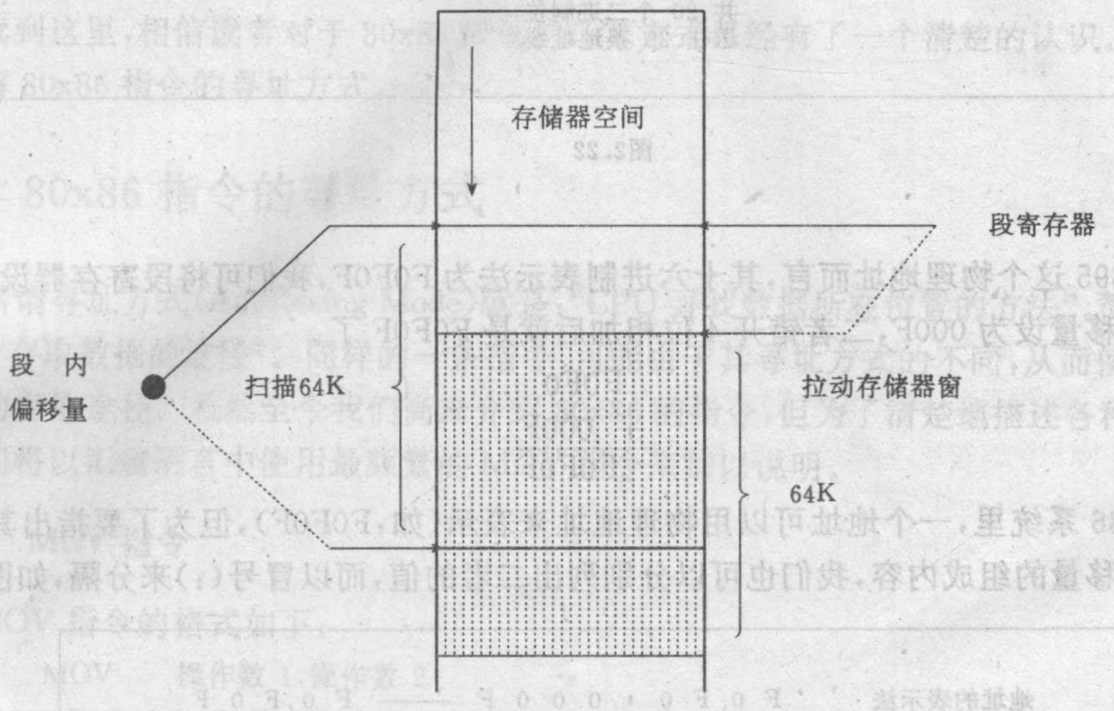


图2.21 存储器窗

如果将一个段看成一个局部存储器空间的话，则段内偏移量就是此局部存储器空间内的地址，而段内偏移量与段寄存器相加所得到的地址就是物理地址了。

2.4.3 逻辑地址与物理地址的存贮空间对照关系

虽然说分段方式可以使得存储器得到有效且结构化的管理，但事实上硬件的 RAM 仍然是由 IC 解码而呈线性的排列。像 8088/86 的 1MB 的 RAM 空间是如何用"段地址:段内偏移量"这种寻址方式表示的呢？

原来，80x86 是将段寄存器的值看成是物理地址中较高的 16 位，即 b4～b19(由 b0 算起)，而将段内偏移量看成是物理地址中较低的 16 位，即 b0～b15。将这两个地址错开 4 个二进制位相加后就造出一个 20 位长的地址来，共可对应到 2^{20}＝1MB RAM 的物理地址了。见图 2.22。

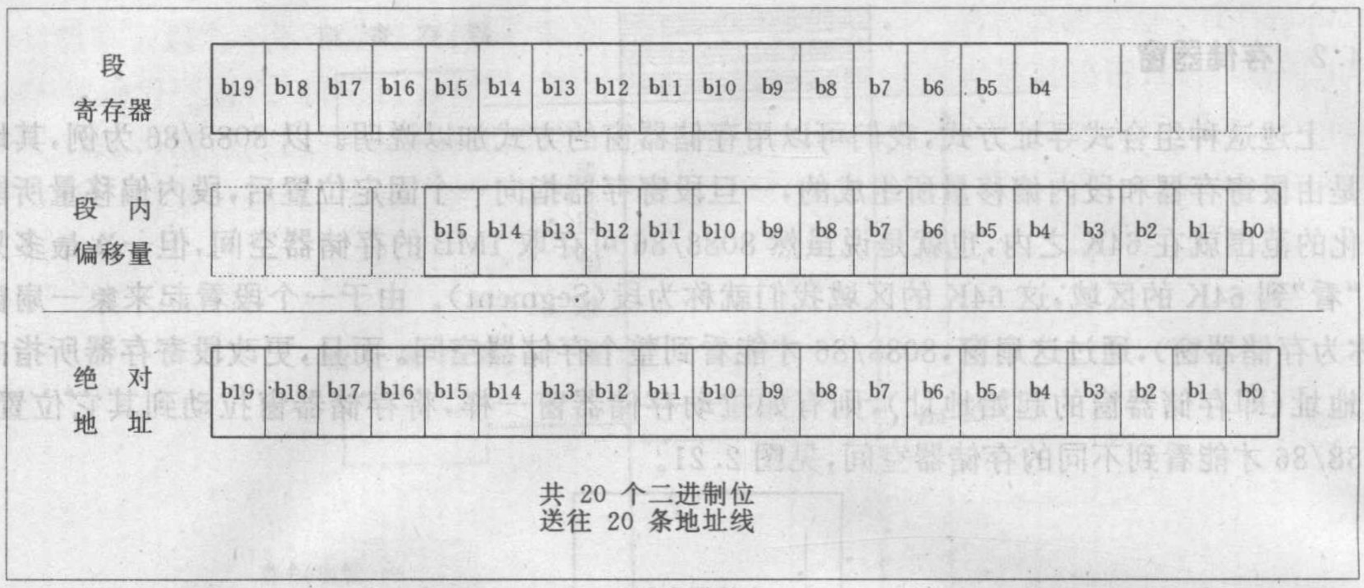


图2.22

以 986895 这个物理地址而言，其十六进制表示法为 F0F0F，我们可将段寄存器设为 F0F0 而将段内偏移量设为 000F，二者错开 4 位相加后就是 F0F0F 了。

$$
\begin{array}{r}
\text{F0F0}\phantom{\text{0}} \\
+\ \ \text{000F} \\
\hline
\text{F0F0F}
\end{array}
$$

在 80x86 系统里，一个地址可以用物理地址来表示（如：F0F0F），但为了要指出其段寄存器与段内偏移量的组成内容，我们也可以分别列出二者的值，而以冒号（:）来分隔，如图 2.23。

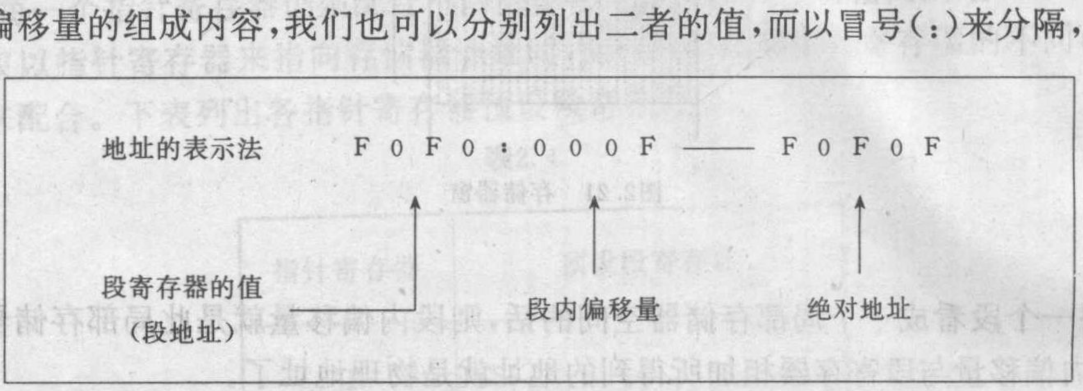


图2.23　80x86 系统中地址的表示法

虽然 F0F0:000F 组合成的地址是 F0F0F，但反过来 F0F0F 却不一定是由 F0F0:000F 组合成的，它可能是由 F000:0F0F 或 F001:0EFF……等非常多的方式组成，除非查出寄存器中真正的内含值，否则我们无法断定一个地址是由什么样的两个值组成的，这一点要加以注意。

一个地址可以用其物理地址来表示，或是用段地址和段内偏移量的组合来表示。而段地址可以直接明指出来，也可以用一个段寄存器的名字来代表，当用段寄存器表示时，其段地址就是此寄存器的内含值，请参考图 2.24 地址的三种表示法。

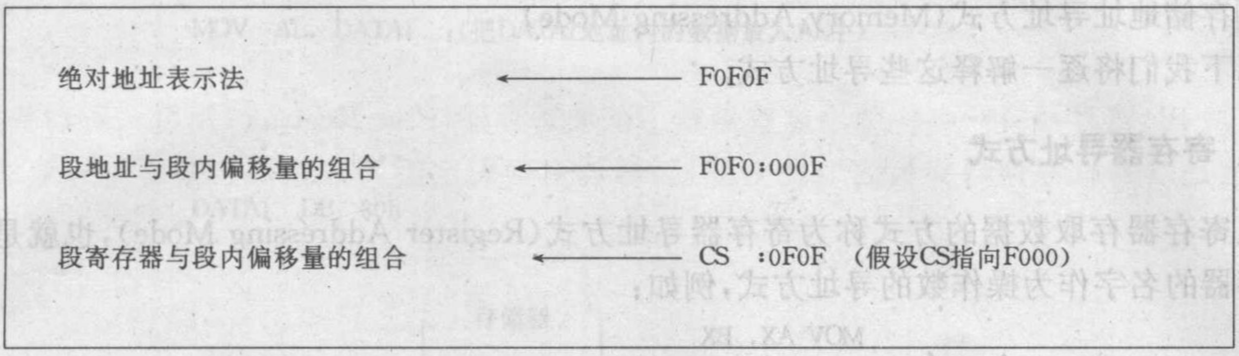


图2.24 F0F0F 的三种表示法

读到这里，相信读者对于 80x86 地址的计算方式已经有了一个清楚的认识。最后，我们要来了解 80x86 指令的寻址方式。

2.5 80x86 指令的寻址方式

所谓寻址方式(Addressing Mode)就是："CPU 寻找数据所在位置的方法"。简而言之，就是"CPU 存取数据的途径"。同样的一条指令，可能由于其寻址方式的不同，从而使得 CPU 采用不同的存取途径。虽然至今我们尚未介绍 80x86 的指令，但为了清楚地描述各种寻址方式，下面我们将以汇编语言中使用最频繁的 MOV 指令来加以说明。

2.5.1 MOV 指令

MOV 指令的格式如下：

MOV　操作数 1，操作数 2

MOV 指令会把参数 2 的数据传送到参数 1 中，传送后参数 1 的内含值（数据）便与参数 2 完全一样。

其中参数 1 和参数 2 可以是寄存器、立即数或存储器地址。请参考以下说明：

□寄存器：以寄存器的名称表示，即 AX，BX，CX，DX 等。

例如　MOV　AX，BX　　;将 BX 的值存入 AX

□立即数：立即数就是一般的常数，可以是二进制数、十进制数或十六进制数，如 00001111B、10、4CH 等。

例如　MOV　AX，260　　;将 260 存入 AX

□存储器地址：当使用存储器地址做为参数时，CPU 便会从所指向的存储器地址中取得数据。

例如　MOV　AX，DATA1　　;将 DATA1 存储器地址中所储存的数据存入 AX

2.5.2 寻址方式的种类

由于数据可以用三种方式储存：存于寄存器、立即数、存储器地址，因此 80x86 的寻址方式也可以分成以下三种：

□寄存器寻址方式(Register Addressing Mode)

□立即寻址方式(Immediate Addressing MOde)

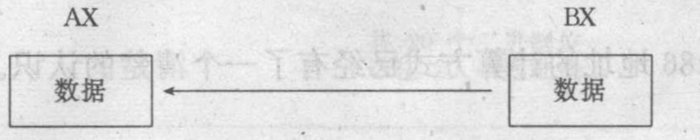
□存储地址寻址方式(Memory Addressing Mode)

以下我们将逐一解释这些寻址方式。

2.5.3 寄存器寻址方式

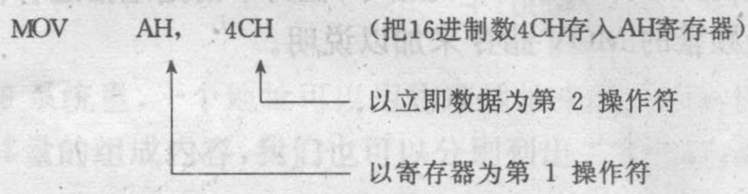
从寄存器存取数据的方式称为寄存器寻址方式(Register Addressing Mode),也就是直接以寄存器的名字作为操作数的寻址方式,例如:

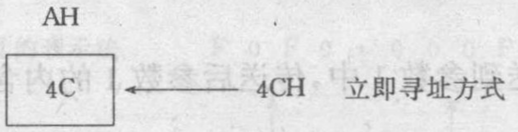
MOV AX, BX



2.5.4 立即寻址方式

立即寻址方式的数据储存于指令本身,由于常数数据就储存在指令里面,所以 CPU 取得该数据的途径是"立即"式的,因此我们把这种寻址方式称为立即寻址方式(Immediate Addressing MOde),立即寻址方式的数据为指令的一部分。例如:





MOV 含有两个操作数,其中只有第二个操作数才可以存放立即数据,而第一个操作数必须是寄存器或存储器地址。

2.5.5 存储地址寻址方式

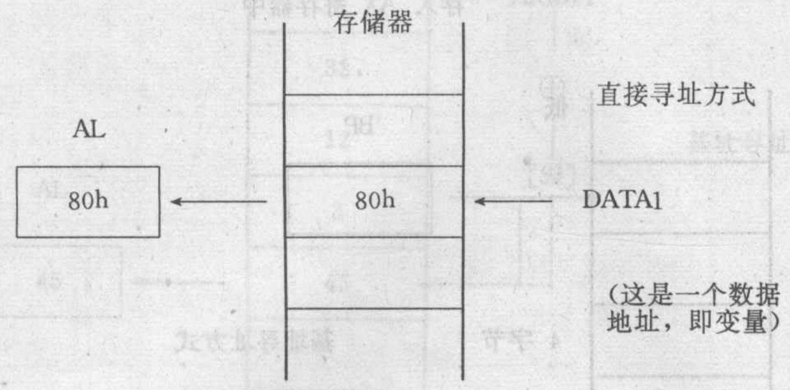
由存储器地址存取数据的方式,称为存储地址寻址方式(Memory Addressing Mode),存储地址寻址方式可细分为以下五种:

<u>直接寻址方式</u>

直接寻址方式(Direct Addressing Mode)就是在指令中直接指出数据所在的地址的方式。例如:

MOV AL，DATA1 ；(把DATA1地址内的数据放入AL中)

DATA1 DB 80h

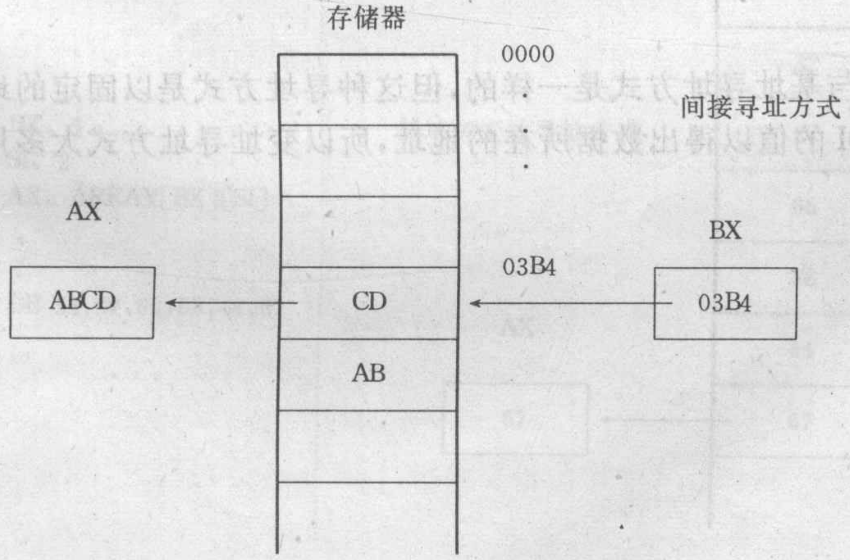


间接寻址方式

间接寻址方式(Indirect Addressing Mode)是以 80x86 中的 BX、BP、SI、DI 寄存器的值来找出数据的地址，再指示 CPU 到该地址存取数据。在间接寻址方式的指令中，寄存器名必须以中括弧[]括起来。例如：

MOV AX，[BX] ← 到BX所指示地址取数据并存入 AX

以 [BX] 寄存器间接寻址

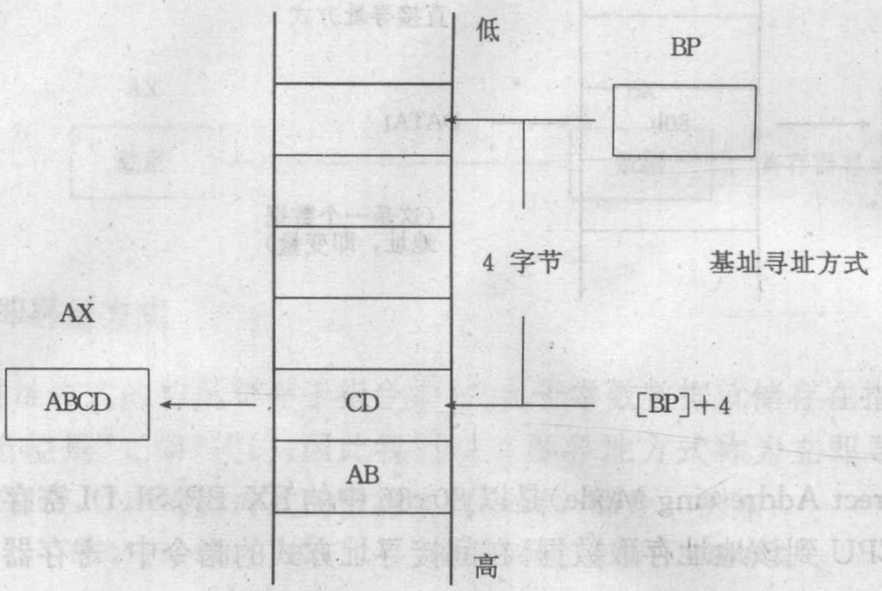


在上例中，BX 寄存器存放着 03B4h 这个地址，当执行指令 MOV AX，[BX]时，CPU 会到 BX 寄存器中取出数据所在的地址(即 03B4h)，然后再到该地址存取数据。

基址寻址方式

BX 及 BP 寄存器有一项特殊用途就是做为基址寄存器(Base Register)使用。所谓基址寻址方式就是以基址寄存器所指的地址为基址，再加上一固定距离来获得数据所在的地址，例如：

MOV AX, [BP]+4 ← 把 BX+4 所指示地址中的内容存入 AX 寄存器中

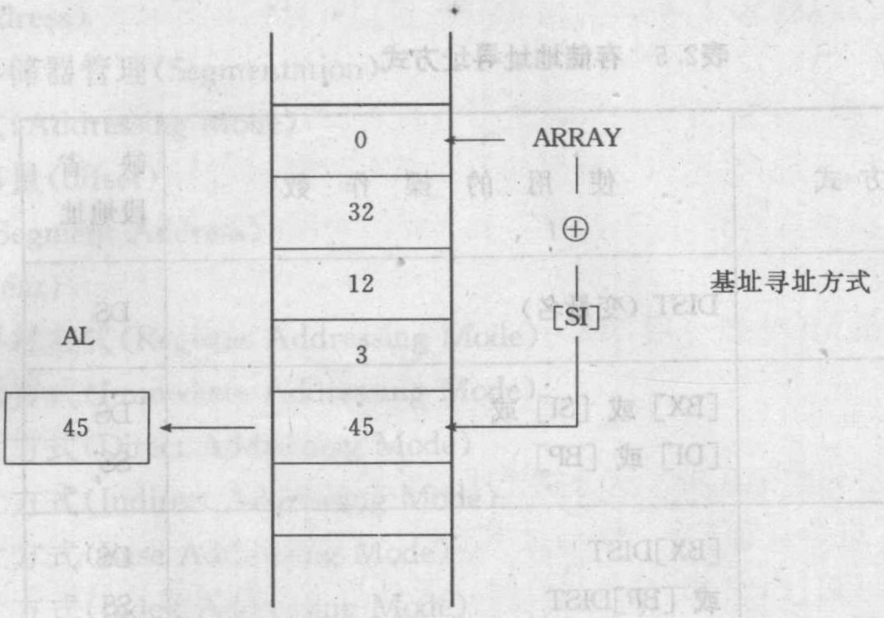


因为以 BP 为基址时，CPU 会将 SS 的值作为其段地址，即缺省段地址，所以 BP 通常用来存取堆栈数据；而以 BX 为基址时，CPU 会将 DS 的值作为段地址，因此 BX 常用来存取一般性的数据。

变址寻址方式

变址寻址方式的原理与基址寻址方式是一样的，但这种寻址方式是以固定的地址为基址再加上变址寄存器 SI 或 DI 的值以得出数据所在的地址，所以变址寻址方式大多用来存取数据阵列中的元素。例如：

```
MOV    SI, 4
MOV    AL, ARRAY[SI]   ← 就是把 ARRAY 地址加上 SI 的值而得出数据所在的地址
    .
    .
ARRAY DB 0,32,12,3,45
```



使用 DI 和 SI,CPU 会配以 DS 作为其段地址。

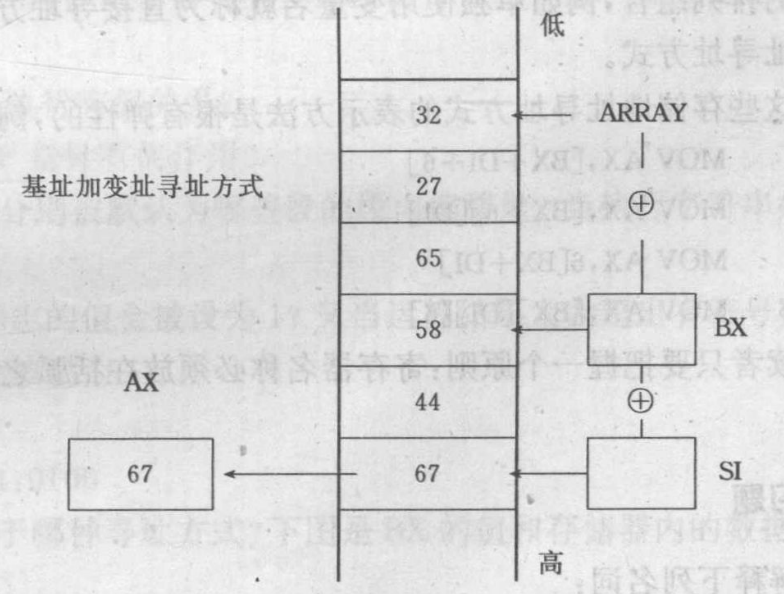
基址加变址寻址方式

把基址寻址方式和变址寻址方式结合起来就是所谓的基址加变址寻址方式,这种寻址方式可以用来存取二维数据阵列元素,例如:

```
MOV  AX,Array[BX][SI]
```

就是把 Array 的起始地址加上 BX 和 SI 这两个索引值来计算出数据所在的地址,例如:

```
MOV    BX, 3
MOV    SI, 2
MOV    AX, ARRAY[BX][SI]
    .
    .
ARRAY DB 32,27,65,58,44,67
```



当使用 BX 为基址时,CPU 会以 DS 为段地址,而使用 BP 为基址时,则 CPU 会以 SS 为段地址。

2.5.6 各种存储地址寻址方式的整理

为了让读者能够更容易分辨各种存储地址寻址方式之间的差异，我们将这些寻址方式整理成表，见表 2.5。①②

表2.5　存储地址寻址方式

寻址方式	使用的操作数	缺省段地址
直接寻址	DIST（变量名）	DS
间接寻址	[BX] 或 [SI] 或 [DI] 或 [BP]	DS SS
基址寻址	[BX]DIST 或 [BP]DIST	DS SS
变址寻址	DIST[SI] 或 DIST[DI]	DS
基址加变址寻址	DIST[BX][DI] 或 DIST[BX][SI] DIST[BP][DI] 或 DIST[BP][SI]	DS SS

如果仔细观察，我们会发现各类存储地址寻址方式实际上就是各个基址、变址寄存器及变量名的排列组合，例如单独使用变量名就称为直接寻址方式，而使用基址寄存器与变量名则称为基址寻址方式。

这些存储地址寻址方式的表示方法是很有弹性的，例如下面的四各写法都是相同的：

```
MOV AX,[BX+DI+6]
MOV AX,[BX+6][DI]
MOV AX,6[BX+DI]
MOV AX,[BX][DI][6]
```

读者只要把握一个原则：寄存器名称必须放在括弧之内，而且各元素必须以“[]”或“+”分开。

本章习题

1. 试解释下列名词：

数据总线(Data Bus)

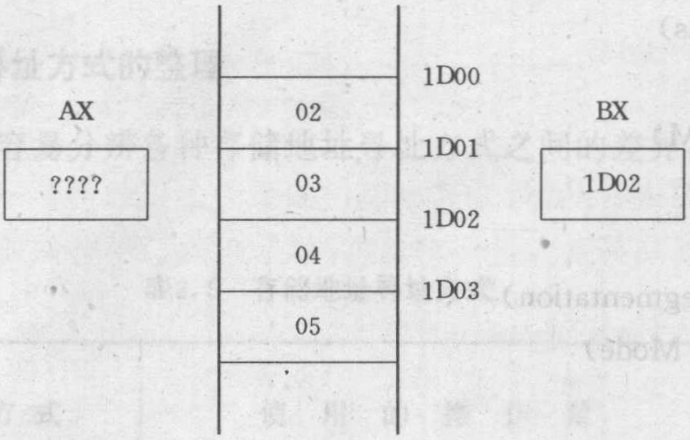
---

① 除了直接寻址方式中的DIST是指变量名以外，其余寻址方式中的DIST可以是变量名、或常数。

② 在基址加变址寻址方式中的DIST可以省略，而仅以基址和变址寄存器来寻址。

地址总线(Address Bus)
寄存器(Register)
随机读写存储器(RAM)
只读存储器(ROM)
地址(Address)
分段式存储器管理(Segmentation)
寻址方式(Addressing Mode)
段内偏移量(offset)
段地址(Segment Address)
段(Segment)
寄存器寻址方式(Register Addressing Mode)
立即寻址方式(Immediate Addressing Mode)
直接寻址方式(Direct Addressing Mode)
间接寻址方式(Indirect Addressing Mode)
基址寻址方式(Base Addressing Mode)
变址寻址方式(Index Addressing Mode)
基址加变址寻址方式(Base Plus Index Addressing Mode)
堆栈指针寄存器(Stack Pointer,简称为 SP)
基址指针寄存器(Base Pointer,简称为 BP)
指令指针寄存器(Instruction Pointer,简称为 IP)
源变址寄存器(Source Index,简称为 SI)
目的变址寄存器(Destination Index,简称为 DI)

2. 请问 PC 是由哪三个基本部分构成的?
3. 试举出 80x86 的通用寄存器有哪些?指针寄存器有哪些?段寄存器有哪些?什么是标志寄存器?
4. 试说明 CS 寄存器、IP 指针和程序的执行有何关系?
5. 试说明什么是堆栈?SS 寄存器和 SP 指针有何作用?
6. 当没有指明段寄存器时,BX、SI、BP 分别被默认为哪些段的段内偏移量?当执行字符串处理指令时 DI 代表哪个段的段内地址?
7. 试问当运算的结果为 0 时,哪一个标志的值会被设为 1?又当运算结果太大超出有符号数所能容纳的范围时,哪一个标志的值会被设为 1?
8. 请计算下列组合地址的物理地址?
   (1) A400:0000　(2) 1B34:0100
9. 请说明 MOV AX,[BX]这条指令属于哪种寻址方式?下图是 BX 的值和存储器内的数据,请问执行指令后 AX=?



10. 请问执行完下列两条 MOV 指令后，AX=？

MOV　BX,10H

MOV　AX,BX

11. 80X86 的寻址方式可分成哪几种？请分别说明其功能。
12. 请问 MOV AX,BX 指令是属于哪一种寻址方式？
13. 当使用 MOV AX,DIST[BP][SI]这条指令时，AX 所取得的数据是来自那个段？它的段内偏移量是多少？（假设 DIST=100h,BP=02h,SI=04h）

第二篇　必备的工具

第三章　如何使用调试程序 DEBUG

DEBUG 程序具有跟踪、调试功能，并且能够汇编简单的汇编语言程序，因此成为学习汇编语言的必要工具之一。本章除了介绍 DEBUG 的使用方法之外，为了配合例题讲解，在第五节中我们还会介绍一些常用的中断服务功能。在本章结束后，读者应学到：

□查看、修改存储器中的内容

□直接输入、执行、跟踪汇编语言程序

□利用 DEBUG 读写磁盘上的数据或程序

□了解一些常用的中断服务程序

为了使读者能顺利阅读往后各章节的内容，首先我们必须说明一些命令参数的表示方式：

□在命令格式中，中括号[]内的项目代表命令的参数，可由使用者根据情况选用，不管这些参数使用与否，命令均会被接受，不过功能却会有所不同。

□如果命令参数中有"甲|乙"表示可以使用甲或乙，但二者只能选一。

3.1　进入及退出 DEBUG

在 MS-DOS 的提示符下键入 DEBUG 命令，即可进入 DEBUG，其命令的格式如下：

DEBUG　[文件全名[参数]]

使用 DEBUG 时，若是在命令后加上文件(或程序)全名(例如：DEBUG EDIT. COM)，则 DEBUG 会将此文件(或程序)载入存储器中供我们进行跟踪和调试。如果所载入的程序还要指定参数，也可以在 DEBUG 命令中一并指定，例如 DEBUG CHKDSK. EXE /F，其中/F 就是 CHKDSK. EXE 程序的参数。

如果不指定文件或程序名称，则执行 DEBUG 后，提示字符会变为"-"，表示已进入 DEBUG 的命令状态，在此状态下我们可以对 DEBUG 下达命令，而且所有的数字都是以十六进制表示的。现在我们先以一个例子来说明 DEBUG 的一般使用方法：

```
C:\DOS>DEBUG [Enter] ←———— 执行 DEBUG       ——— 显示各寄存器内容
-R ←———————————————— 键入 [R] 后按 [Enter]       ↓
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=2042  ES=2042  SS=2042  CS=2042  IP=0100   NV UP EI PL NZ NA PO NC
2042:0100 0F                DB    0F
-Q ←———————————————— 键入 [Q] 后按 [Enter]
C:\DOS> ←——————————— 结束 DEBUG，回到 DOS
```

在这个例子里，我们使用 DEBUG 来查看寄存器中的内容。其中 R 是 DEBUG 的 Register 命令，R 命令会以十六进制数显示目前 CPU 各寄存器的内容，所以我们在屏幕上可以看到各

寄存器的初始值。最后我们以 Q(Quit)命令结束 DEBUG,回到 DOS。

3.2 计算十六进制数

我们可以使用 DEBUG 的 H 命令(Hexadecimal 的意思)来计算十六进制数:

H(ex)
用　途:以十六进制方式计算两个数的和与差
格　式:H　数值1　数值2

H 命令会以十六进制方式算出两个数的和与差,并且显示到屏幕上。所有数字都以十六进制表示,最长可输入 4 位数。

实　例

```
- H4538  5623
9B5B  EF15
 ↑     ↑
 |     └──── 4538 与 5623 之差(有借位, 为负值)
 └────────── 4538 与 5623  和
```

3.3 查看存储器的内容

我们可以使用 DEBUG 的 D 命令来查看存储器的内容。

D(ump)
用　途:显示指定的存储区内容
格　式:D [范围]

D 命令可以倾卸(Dump)出所指定的存储器内容,并且分别以十六进制值及 ASCII 字符显示出来,如果其值在控制字符范围内(0~1Fh)或高于 127(高于 7Fh,为 ASCII 码以外的范围),则用“.”表示,因为这些范围内并无 ASCII 可见字符。

实　例

用 D 命令列表显示时,所显示的第 8 至第 9 个字节间有一个连字符“-”连接,以便于查看,比如:

```
-D00,7F ←—显示 00 到 7F 之间的存储区
29B7:0000  CD 20 00 A0 00 9A EE FE-1D F0 F4 02 08 26 2F 03   ............&/.
29B7:0010  08 26 BC 02 08 26 DC 0C-01 03 01 00 02 FF FF FF   .&....&........
29B7:0020  FF FF FF FF FF FF FF FF-FF FF FF FF BF 0D 4E 01   ..............N.
29B7:0030  42 29 14 00 18 00 B7 29-FF FF FF FF 00 00 00 00   B).....)........
29B7:0040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
```

```
29B7:0050  CD 21 CB00  00 00  00  00-00  00 00 00 00 20 20 20   .!..........
29B7:0060  20  20 20 20  20 20  20  20-00  00 00 00 00 20 20 20   .....
29B7:0070  20  20 20 20  20 20  20  20-00  00 00 00 00 00 00 00   ........
-
```

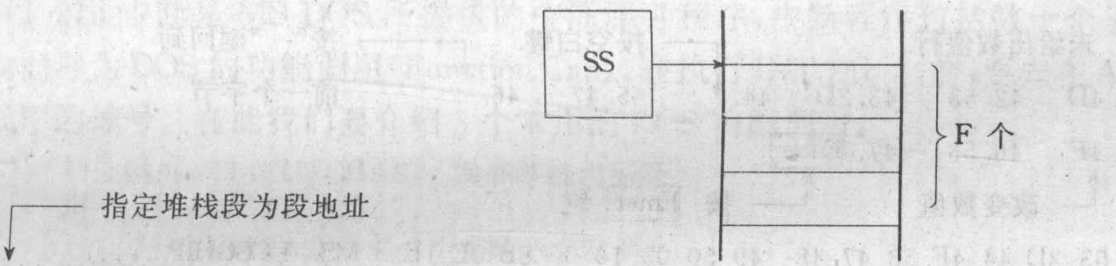
上图就是用“D”命令倾卸存储器中的内容的格式,其中“xxxx:xxxx”是以段地址和段内偏移量的组合形式表示的存储器地址,每行显示 16 个字节,第 8,9 字节之间的一个连字符“-”,在连字符的左右各有 8 个字节。

如果没有指定倾卸的范围,则 DEBUG 会从上一次 D 命令所显示的最后地址,接着显示 8 行,每行 16 个字节,共 128 个字节。

```
-D00,0F
29B7:0000  CF 90 00 A0 00 9A EE FE-1D F0 F4 02 08 26 2F 03   ............&/.
-D
29B7:0010  08  26 BC 02  08 26  DC 0C-01  03 01 00 02 FF FF FF   .&...&..........
29B7:0020  FF FF FF FF  FF FF  FF  FF-FF  FF FF FF BF 0D 4E 01   ..............N.
29B7:0030  42  29 14 00  18 00  B7  29-FF  FF FF FF 00 00 00 00   B).....)........
29B7:0040  00  00 00 00  00 00  00  00-00  00 00 00 00 00 00 00   ................
29B7:0050  CD 21 CB00  00 00  00  00-00  00 00 00 00 20 20 20   .!..........
29B7:0060  20  20 20 20  20 20  20  20-00  00 00 00 00 20 20 20   .....
29B7:0070  20  20 20 20  20 20  20  20-00  00 00 00 00 00 00 00   ........
29B7:0080  00  0D 6D 31  00 0D  1E  B1-42  B8 FF FF 87 06 F5 0B   ..m1....B.......
-
```

我们也可以用‘L’来指定倾卸的长度,D100 L20,即是由 DS:100 开始,列表显示出 20h 个字节,其效果等同于 D100,11F。

在 D 命令中,如果不指出段地址,则 DEBUG 会把 DS 的值缺省为段地址,亦即 D00,7F 会列出 DS:00~DS:7F 这一区域中的值。当然我们也可以通过设置指定其它的段地址。




```
- dSS:0 L0F
2042:0000 CD 20 00 A0 00 9A 4E 4F- 17 FB FA 02 CF 95 35   . ....NO.....5
-
```

如果读者也尝试着在自己的电脑上执行这个例子,其倾卸的内容与这里所列的可能会有不同,这是因为段地址不同或者是电脑存储器内的数据不同的缘故。

3.4 将数据输入存储器

我们可以使用 DEBUG 的 E 命令(即 Enter 的意思),将数据输入存储器。

E(nter)	
用　途:输入数据 格　式:E 起始地址 [数据行]	

E 命令有下列两种使用方式:

1)如果命令中含有数据行,则数据行就会输入到指定的地址上去。

实例

```
  ┌── 起始地址
- E100 41 42 43 44 45 46 47 48 49 50        ←── 数据行
- D100,10F                                  ←── 显示 100 到 10F 的内容
2042:0100  41 42 43 44 45 46 47 48- 49 50 05 10 00 8B 0E 1E   ABCDEFGHIP......
- E100 "PIHGFEDCBA"                         ←── 字符串
- D100,10F                                  ←── 显示 100 到 10F 的内容
2042:0100  50 49 48 47 46 45 44 43- 42 41 05 10 00 8B 0E 1E   PIHGFEDCBA......
```

2)如果 E 命令只指定起始地址而没有数据行,则 DEBUG 会显示出该地址及其内容,此时使用者会先看到每一个字节的内容,然后再输入新数据。如果使用者输入减号"-",则会回到前一个字节上,同时以此字节为新的起始地址。人们可以用空白键来表示要保留目前字节的值并跳到下一地址。如果想中止 E 状态,则只要按下 Enter 就可回到 DEBUG 的命令状态。

实例

```
- E100 ←── 未给出数值行          ┌── 按空白键       ┌── 按"- "键回到
2042:0100   41.4D   42.53   43.2D   44.       45.47   46.-      前一个字节
2042:0104   47.4F   46.53   47. Enter
- D100        └── 改变数值        └── 按 Enter 键
2042:0100  4D 53 2D 44 4F 53 47 48- 49 50 05 10 00 8B 0E 1E   MS- DOSGHIP......
2042:0110  ......................................................  ┐
2042:0120  ......................................................  │
2042:0130  ......................................................  │
2042:0140  ......................................................  ├(略)
2042:0150  ......................................................  │
2042:0160  ......................................................  │
2042:0170  ......................................................  ┘
-
```

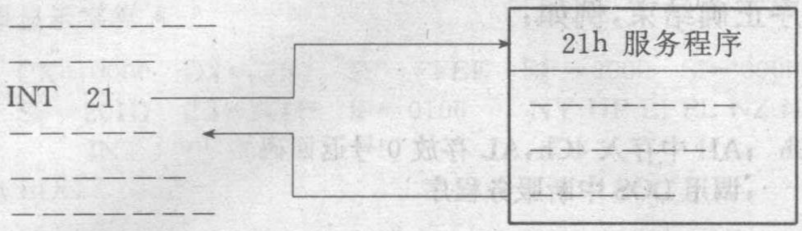
如果我们在 E 命令中没有指定段地址,则 DEBUG 把 DS 寄存器中的地址作为缺省的段地址。

3.5 中断原理的说明

所谓中断，就是指 CPU 在执行程序的过程中如果接收到中断请求信号，则会暂停目前的工作而去执行中断服务程序。在汇编语言程序中，我们可以用 INT 指令来产生中断，其格式如下：

INT xx

其中 INT 是中断指令，XX 是用来表示所要调用的中断服务程序编号。当中断产生时，CPU 会暂停目前的工作，而直接跳往指定的中断服务程序去执行该服务程序，待执行完后，再自动回到原程序中断的地方继续执行原程序。比如下例就是调用第 21h 编号的服务程序。



注：在DEBUG中所有的数字均为十六进制，因此，INT 21 就是指INT 21h。

大多数的中断服务程序在调用前，还必须将一个或多个参数先存入固定的寄存器中，以便传送所需的信息给服务程序，稍后我们将举例来说明。

3.5.1 INT 21h 中断

INT 21h 中断是 MS-DOS 所提供的中断服务程序，中断程序包括数十个子程序，这些子程序我们称为 DOS 的功能调用(Function Call)。在执行 INT 21h 之前，要先在 AH 寄存器内指定子程序的编号。在此我们要介绍 3 个常用的 DOS 功能调用。

功能调用：02h(PUTCHAR)，向屏幕输出字符
用　　法：AH 存放 02h
　　　　　DL 存入所要显示字符的 ASCII 码
返回参数：AL＝字符键码

02h 功能调用会把 DL 寄存器内的字符送到屏幕上去，例如：

```
MOV AH,02h        ;AH 寄存器存入功能调用编号(第 2 号)
MOV DL,41h        ;DL 寄存器储存“A”的 ASCII 码
INT 21h           ;调用 DOS 中断服务程序
```

该调用会在屏幕上当前光标的位置上显示 A 这个字符。我们稍后将介绍如何在 DEBUG 中输入、执行这个程序，目前读者只要能了解就行了。

功能调用：09h(PRINT)，在屏幕上显示一个字符串
用　　法：AH 存放 09h
　　　　　DS:DX　　指向所要显示字符串的地址
返回参数：无

09h 功能调用会把 DS:DX 所指向的字符串输出到屏幕上，直到遇到“$”字符为止。所以我们必须在字符串的最尾端加上“$”符号，以终止显示输出。例如在 DEBUG 下我们输入以下程序：

```
MOV AH,09h          ;AH 寄存器存入功能调用编号(第 09 号)
MOV DX,109          ;指向字符串起始地址,假设字符串地址在 109
INT 21h             ;调用 DOS 中断服务程序
    .
DB "This is a test. $" ;使用 DB 伪指令定义字符串
```

执行后在屏幕上当前光标处显示"This is a test."信息。[①]

功能调用:04Ch 结束当前进程返回父进程

用　　法:AH 存放 04Ch

　　　　　AL 存放返回码

返回参数:无

4Ch 可用来结束程序的执行,并可由 AL 传回返回码。当程序在 DOS 下执行时,我们可以由批处理文件中的 ERRORLEVEL 命令来获得返回码,以判断程序是在何种情形下结束的,我们通常以 0 代表程序正确结束,例如:

```
............
..........
MOV AX,4C00h  ;AH 中存入 4Ch,AL 存放 0 号返回码
INT 21h       ;调用 DOS 中断服务程序
```

3.5.2 INT 3

主要用于 DEBUG 中让程序停止执行。当程序执行到这个中断指令时会停止并显示各寄存器的内容。

3.6 用 DEBUG 编写汇编语言程序

DEBUG 的 A 命令可以用来编写汇编语言;U 命令可以用来进行反汇编,也就是将程序列表出来;R 命令则可以列出所有寄存器的内容。另外,G 命令则可以执行已编辑好的程序。

3.6.1 A 命令

A(ssemble)	
用　途:将汇编语言指令编译成机器码,并存入存储器地址中。	
格　式:A [起始地址]	

A 命令是用来将所输入的汇编语言指令编译成机器码,并且存入指定的存储器地址。在输入过程中,除了引号括起来的字符串(给 DB 或 DW 用的)外,所有数字均会被当成十六进制数。在 A 后面我们可以指定程序的起始地址,如果第一次下达 A 命令而没有指定地址,则 DEBUG 会缺省 CS:100 做为起始地址。但第二次以后如果不指定地址,则会从上次 A 命令的结束地址开始。

① 由于 DEBUG 缺省 DS 和 CS 指向同一个段,因此在此例中 DS 实际上已指向字符串所在段,所以就不需要再设定 DS 的值了。

实　例

```
-A ←————进入汇编状态
              ┌———键入下列汇编语言程序
201D:0100  MOV AH,02  ←———— DOS 的第2号功能调用
201D:0102  MOV DL,41  ←————"A"的 ASCII 码存入 DL 寄存器
201D:0104  INT  21  ←———调用中断程序
201D:0106  [Enter]  ←————离开 A 状态
-A ←————啊!忘了输入最后一行程序,再进入汇编状态
  ┌———自动设定由上次结束地址开始输入
201D:0106  INT  3  ←————中断执行
201D:0107  [Enter]  ←————离开 A 状态
-G ←————执行
A ←————执行结果显示字符 A
AX=0241  BX=0000  CX=0000  DX=0041  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=201D  ES=201D  SS=201D  CS=201D  IP=0106   NV UP EI PL NZ NA PO NC
201D:0106  CC          INT         3
-Q ←————离开 DEBUG
C:\> ←————回到 DOS
```

注:DEBUG 中的缺省数制是十六进制,因此 INT 21就是 INT 21h,其中的 h 不用再输入。

在上述例子中,我们已经输入并执行了一个小程序,是否很方便迅速呢?

在汇编状态下我们可以直接输入80x86的汇编语言指令,比如:MOV、INT、…等,也可以使用简单的伪指令:DB、DW、…等等,对于小型程序来说已经足够应用了。以下为使用 DB 伪指令的例子。

实　例

```
C:\>DEBUG
- A ←————————————进入汇编状态
  ┌———————————————表示存储器地址
201D:0100   MOV AH,09 ←———— DOS 的 09 号功能调用, 显示字符
201D:0102   MOV DX,108 ←———— 指向字符串起始地址
201D:0105   INT 21 ←———————— 调用中断服务程序
201D:0107   INT 3
201D:0108   DB "HELLO MASM611! $" ←——— 使用 DB 伪指令定义字符串
201D:0117   └—— 伪指令
- G ←————————————执行此程序
HELLO MASM611! ←————显示执行结果
AX=0924  BX=0000  CX=0000  DX=0108  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=201D  ES=201D  SS=201D  CS=201D  IP=0107   NV UP EI PL NZ NA PO NC
201D:0107   CC          INT      3
-
```

何谓伪指令

在上述程序中，MOV、INT 都是80x86 CPU 的指令，但 DB 却不是 CPU 的指令。DB、DW 这些指令是用来告诉 DEBUG：在双引号内的数据是要用字节或字的方式来存放。DEBUG 会按照 DB 或 DW 所指定的方式去存取双引号内的数据，然后将 DB、DW 这些字丢弃，当 CPU 真正执行程序时是看不到 DB、DW 的，因此称为伪指令。

3.6.2 U 命令

U(nassemble)	
用　途：将存储器内的机器码反汇编成汇编语言指令 格　式：U［存储器范围］或［地址］	

U 命令会将指定的存储器内的机器码转换成汇编语言，并且连同该指令的地址及机器码一起显示出来，由于每个指令的长度不一(1个或多个字节)，所以显示出来的存储器区域可能比指定的(或系统缺省的)要多。

所指定的起始地址必须正确指向指令的第一个字节，否则会被错误的反汇编。

U 命令有两种使用方式，现说明如下：

1)不指定反汇编的范围，则每一次 U 命令会由指定的地址开始反汇编32个字节，命令格式如下：

U ［地址］

上述的地址参数可以是“段内偏移量”、“段地址：段内偏移量”、或省略不写，如果仅指定段内偏移量，则 DEBUG 会缺省 CS 为段地址，如果指定了段地址与段内偏移量，则 DEBUG 会由该地址开始反汇编，如果第一次使用 U 命令而未指定地址，则 DEBUG 会由目前 CS：IP 的地址开始反汇编，尔后每一次使用 U 命令而不指定地址时，DEBUG 会自动由上一次 U 命令之后的地址开始反汇编。

实 例

```
C:\DOS>DEBUG CHKDSK.EXE
- R
AX=0000  BX=0000  CX=2DD1  DX=0000  SP=0200  BP=0000  SI=0000  DI=0000
DS=2042  ES=2042  SS=2350  CS=2042  IP=0100   NV UP EI PL NZ NA PO NC
2042:0100   B83E06          MOV     AX,063E
- U 100 ←—— 仅指定段内偏移量(100 也可省略，因为 IP 的预设值为 100h)
  └———————— CS 的值
2042:0100   B83E06          MOV     AX,063E      ┐
2042:0103   BAF602          MOV     DX,02F6      │
2042:0106   055220          ADD     AX,2052      │
2042:0109   3B060200        CMP     AX,[0002]    │
2042:010D   721B            JB      012A         │  反
2042:010F   B409            MOV     AH,09        │  汇
2042:0111   BA1801          MOV     DX,0118      │  编
2042:0114   CD21            INT     21           │  的
2042:0116   CD20            INT     20           │  结
2042:0118   4E              DEC     SJ           │  果
2042:0119   6F              DB      6F           │
2042:011A   7420            JZ      013C         │
2042:011C   65              DB      65           │
2042:011D   6E              DB      6E           │
2042:011E   6F              DB      6F           │
2042:011F   7567            JNZ     0118         ┘ 从 100～11F 共 32 个字节
    地址       机器码           汇编语言指令
```

2)指定反汇编的范围

U 起始地址 结束地址(或 L 长度)

如果指定了范围,则整个范围内的字节均会被反汇编。

```
-U 0 6 ←——————反汇编0～6这个范围
201D:0000 CD20          INT     20
201D:0002 FF9F009A      CALL    FAR [BX+9A00]
201D:0006 EE            OUT     DX,AL
-U 7 L4←———————————由地址0007开始反汇编4个字节
201D:0007 FE1D          CALL    FAR [DI]
201D:0009 F0            LOCK
201D:000A 4F            DEC     DI
-
```

3.6.3 R 命令

R(egister)	
用　途:显示及更改寄存器的值 格　式:R [寄存器名]	

此指令会以十六进制数显示 CPU 各寄存器当前的内容。

实　例

```
- R
AX=0000  BX=0000  CX=2DD1  DX=0000  SP=0200  BP=0000  SI=0000  DI=0000
DS=2042  ES=2042  SS=2350  CS=2042  IP=0100   NV UP EI PL NZ NA PO NC
2042:0100   B83E06        MOV   AX,063E
```

如果在 R 指令之后指定寄存器名称，则 DEBUG 会单独显示此寄存器的内容，并出现“:”符号，等待我们输入新的数值，如果我们不想改变此寄存器的值，只要按下 Enter 键即可。正确的寄存器名称如下：

AX，BX，CX，DX，SP，BP，SI，DI，DS，ES，SS，CS，IP，F

实　例

```
- R
AX=0000  BX=0000  CX=2DD1  DX=0000  SP=0200  BP=0000  SI=0000  DI=0000
DS=2042  ES=2042  SS=2350  CS=2042  IP=0100   NV UP EI PL NZ NA PO NC
2042:0100   B83E06        MOV   AX,063E
- RAX
AX 0000        ;原来的 AX 值
:345 <------- 改变 AX 寄存器的值
- RIP
IP 0100
:105                                   寄存器改变后的值
- R
AX=0345  BX=0000  CX=2DD1  DX=0000  SP=0200  BP=0000  SI=0000  DI=0000
DS=2042  ES=2042  SS=2350  CS=2042  IP=0105   NV UP EI PL NZ NA PO NC
2042:0105   055220        ADD   AX.2052
- RSX
br Error <------ 错误的寄存器名称
-
```

如果我们想改变标志寄存器的值（输入 RF 命令），则 DEBUG 会先将标志寄存器中的内容显示出来，如果我们想改变某个标志位，只需输入该标志位的名称即可。表3.1列出了各标志位在被设置（为1）或被清除（为0）时的表示法。

表3.1 各个标志位的表示法

标 志 名 称	设置（SET）	清除（CLEAR）
OVERFLOW（溢出）	OV（溢出）	NV（未溢出）
DIRECTION（方向）	DN（减少）	UP（增加）
INTERRUPT（中断）	EI（允许）	DI（禁止）
SIGN（符号）	NG（负）	PL（正）
ZERO（零位）	ZR（为零）	NZ（不为零）
AUXILIARY CARRY（辅助进位）	AC（进位设定）	NA（进位清除）
PARITY（奇偶标志）	PE（偶）	PO（奇）
CARRY（进位）	CY（进位设定）	NC（进位清除）

在输入 RF 指令后，则 DEBUG 会将各标志位按照表3.1的次序依次显示在同一行上，并在最后显示一个减号“-”，以供我们输入新的标志位值。在输入时，可以不必按照表3.1的次序，两个标志位之间也可以不留空格，最后按 Enter 键，便可结束。如果输入了错误的标志位设置，或对某个标志位输入了两个以上的新值时，DEBUG 会显示一个出错信息：

df Error

实 例

```
- R
AX=0000  BX=0000  CX=2DD1  DX=0000  SP=0200  BP=0000  SI=0000  DI=0000
DS=2042  ES=2042  SS=2350  CS=2042  IP=0100   NV UP EI PL NZ NA PO NC
2042:0100   B83E06          MOV     AX,063E
- RF
NV UP EI PL NZ NA PO NC   - CYZRPE  ←———— 输入标志名
- R
AX=0345  BX=0000  CX=2DD1  DX=0000  SP=0200  BP=0000  SI=0000  DI=0000
DS=2042  ES=2042  SS=2350  CS=2042  IP=0100   NV UP EI PL ZR NA PE CY
2042:0100   B83E06          MOV     AX,063E                    ↑
- RF                                                      标志值改变了
NV UP EI PL ZR NA PE CY   - UP DN  ←———— 对同一标志输入两个新值
df Error ←———————————————— 出错信息
-
```

3.7 程序的存盘与载入

在 DEBUG 中,我们可以先用 N 命令指定文件名,接着使用 L 命令将磁盘中的文件载入存储器,或使用 W 命令将存储器中的内容存入磁盘文件中。

3.7.1 N 命令

N(ame)	
用　途:设定要载入存储器(或写入磁盘)的文件名或程序执行时的参数 格　式:N [文件名 | 参数]	

在第1节中我们曾经提到,在进入 DEBUG 时可以同时指定所要调试的程序名称及该程序的参数。例如:

```
                                  ┌参数 (EDIT 在执行时要读取的文件)
C:\DOS>DEBUG EDIT.COM FILE1
                 └欲调试的程序 (EDIT.COM 是一个文字编辑程序)
```

N 命令则可以让我们进入 DEBUG 后,再设定所要载入的程序名称、执行程序的参数、或者是要写入磁盘的文件名称。要达到上述功能,N 命令必须分别配合 L(载入文件)、G(执行程序)、或 W(写入磁盘)等命令:

□N 命令后接着执行 L 命令:表示用 N 命令设定文件名,再将文件载入存储器。

□N 命令后接着执行 G 命令:表示用 N 命令设定参数,再执行 DEBUG 中的程序。

□N 命令后接着执行 W 命令:表示用 N 命令设定文件名,再将文件写入磁盘中。

为了便于解说,我们将在稍后配合 L 及 W 命令举例来说明 N 命令的用法。

3.7.2 L 命令

L(oad)	
用　途:将文件中的数据载入存储器 格　式:L 载入地址	

使用 L 命令前,我们必须事先以 N 命令指定文件名,然后指定欲载入文件的地址。

所设定的载入地址是指段内偏移量,系统会自动的配以 CS 作为段地址,如果不设定载入地址,则文件会被载入至 CS:100h。.EXE 与.COM 文件则不受载入地址参数的影响,COM 文件会固定地被载入 CS:100h 地址,而 EXE 文件则会自行设定载入地址。

文件载入后，BX:CX 记录着载入文件的长度。如果载入的是.EXE 文件，则 BX:CX 记录的是程序真正的长度，而不包括.EXE 文件的文件头部分。BX:CX 所代表的是一个4字节长的正整数。假设程序的长度为128K，则：

```
BX=0002h
CX=0h
BX:CX=20000h=128k
```

实　例

```
C:\DOS>COPY CON FILE1 ←—————— 建立一个文本文件
THIS IS FILE1. ←—————— 键入内容
^Z ←—————— 按 CTRL Z 结束并存盘
        1 file(s) copied

C:\DOS>COPY CON FILE2 ←—————— 建立另一个文本文件
THIS IS FILE2.
^Z
        1 file(s) copied
                          ┌———— 用 DEBUG 跟踪 EDIT.COM
C:\DOS>DEBUG EDIT.COM FILE1 ←—————— EDIT.COM 的参数
-G ←—————— 执行（显示EDIT.COM的画面后再执行
                   File 菜单中的 Exit命令）
Program terminated normally ←—————— 程序执行结束
-NEDIT.COM ←—————— 用 N 命令重新设定载入程序的名称
-L ←—————— 载入程序
-NFILE2 ←—————— 用 N 命令设定执行程序的参数
-G ←—————— 执行

Program terminated normally
-
```

所以我们便知道如何使用 N 及 L 命令来载入文件并设定参数了，而储存文件的方法，则是使用 N 命令和 W 命令。

3.7.3　W 命令

W(rite)	
用　途：将程序写入磁盘中的文件 格　式：W 起始地址	

W 是与 L 相反的命令，W 会把存储器中的数据写入磁盘，执行 W 之前要以 N 命令来设定文件名，并指定起始地址。如果省略起始地址，则系统会缺省 CS:100作为起始地址。在存盘时，DEBUG 会由指定的起始地址开始，按照 BX:CX 所设定的长度写入 N 所指定的文件中。

```
C:\>DEBUG
-A
201D:0100 MOV AH,09                     ┐
201D:0102 MOV DX,10B                    │
201D:0105 INT  21                       │ 输入这一段程序
201D:0107 MOV AX,4C00                   │
201D:010A INT  21                       │
201D:010C DB   "This is TEST.COM file. $"┘
201D:0123
-NTEST.COM   ←——————— 设定存盘文件名
-RCX   ←——————————— 设定 CX 寄存器内容
CX 0000
:23   ←————————— 存盘文件长度 (123h-100h=23h)
-W   ←————————— 存盘
Writing 00023 bytes
-Q

C:\DIR TEST.COM   ←——— 看看文件是否存在

Volume in drive C is A A
Volume Serial Number is IE63-8505
Directory of C:\

TEST      COM           35 04-17-95   2:53p  ←—— 果然存在
  1 file(s)             35 bytes  ←——— 长度为 35 (23h)
                  146,096,128 bytes free

C:\>TEST   ←——— 执行看看
This is TEST.COM file.   ←——— 执行结果

C:\>
```

3.8 执行程序

G(o)	
用　途：执行程序	
格　式：G[=起始地址][停止地址]	

G 命令是用来执行 CS:IP 所指向的程序，如果指定了起始地址(=起始地址)，则 IP 会先指向该地址然后开始执行。在下达 G 命令时，务必注意 CS:IP 的值，因为错误的 CS:IP 地址将使 G 命令执行到错误的程序地址，很容易就会死机。在 G 命令中如果没有指定地址，则 DEBUG 会采用当前的 IP 值。

如果在起始地址后同时设定了停止地址，则程序会执行到所指定的地址前停止，然后显示所有寄存器的值和停止地址上的指令。

3.9 跟踪程序

P(roceed)	参考 T 命令
用　途:执行指定地址上的程序，可以指定指令的个数 格　式:P [=地址][指令个数]	

P 命令是用来跟踪(TRACE)程序执行过程的命令。我们可以在 P 命令中指定程序执行的起始地址,以及想要执行的指令个数,如果未指定则一次执行一条指令。

实　例

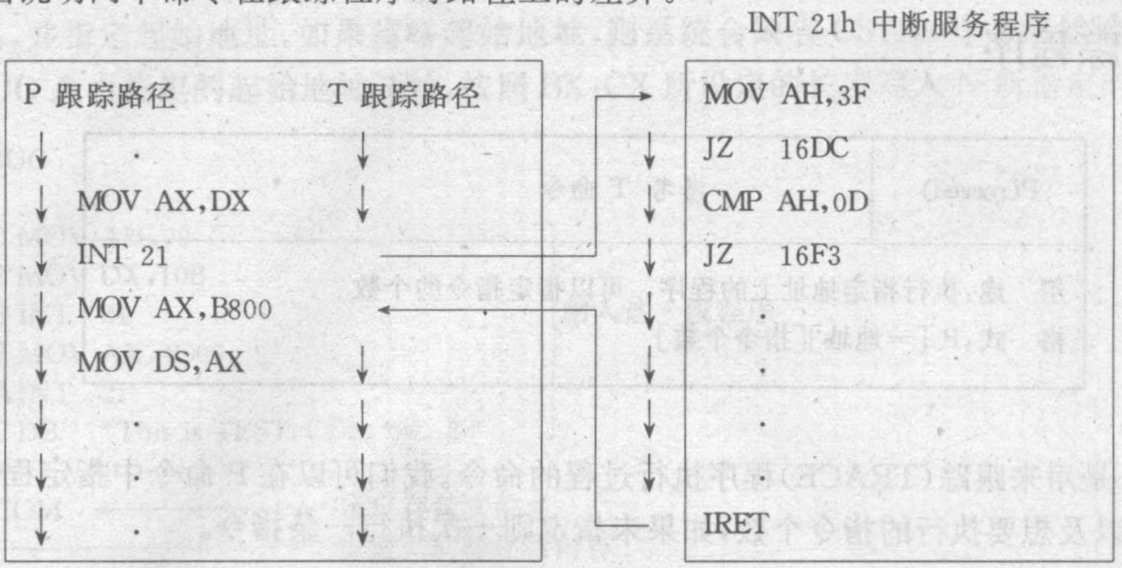
```
C:\DEBUG
-A ←———— 先编写一段小程序
2042:0100  MOV DL,03  ←——  显示"♥"
2042:0102  MOV AH,02
2042:0104  INT  21
2042:0106  INT  3
2042:0107
-P=100 2  ←——由地址100h 开始,执行两个指令
AX=0000  BX=0000  CX=0000  DX=0003  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=2042  ES=2042  SS=2042  CS=2042  IP=0102   NV UP EI PL NZ NA PO NC
2042:0102  B402           MOV      AH,02
AX=0200  BX=0000  CX=0000  DX=0003  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=2042  ES=2042  SS=2042  CS=2042  IP=0104   NV UP EI PL NZ NA PO NC
2042:0104  CD21           INT      21  ←——  已执行两条指令,停在这里
-P  ←————  执行 INT 21 这个指令看看
♥  ————  第 2 号功能调用的执行结果
AX=0203  BX=0000  CX=0000  DX=0003  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=2042  ES=2042  SS=2042  CS=2042  IP=0106   NV UP EI PL NZ NA PO NC
2042:0106  CC             INT      3  ←——  停在这里
-
```

由上例我们可以发现,像 INT 21这种中断指令会被 P 命令当成单一指令来执行,而不会进入服务程序内部对其程序代码做进一步的跟踪。如果想跟踪进入中断服务程序或其它子程序内部,可以使用 T 命令来代替 P 命令。

T(race)	参考 P 命令
用　途:跟踪指定地址上的程序，可以指定指令的个数 格　式:T[=地址][指令个数]	

T 命令与 P 命令相同,也是用来跟踪程序的,差别在于 P 命令把调用子程序指令(CALL)和中断指令(INT)当成是单一指令来执行,而 T 命令则会跟踪进 CALL 或 INT 的子程序内部

细节。下图说明两个命令在跟踪程序时路径上的差异。



P 命令和 T 命令跟踪程序路径的差异

实 例

```
C:\DEBUG
-A ←——  先编写一段小程序
2042:0100  MOV   DL,03  ←——  显示"♥"
2042:0102  MOV   AH,02
2042:0104  INT   21
2042:0106  INT   3
2042:0107
-T=100  2  ←——  由地址 100 开始,执行两条指令
AX=0000  BX=0000  CX=0000  DX=0003  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=2042  ES=2042  SS=2042  CS=2042  IP=0102  NV UP EI PL NZ NA PO NC
2042:0102  B402          MOV     AH,02
AX=0200  BX=0000  CX=0000  DX=0003  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=2042  ES=2042  SS=2042  CS=2042  IP=0104  NV UP EI PL NZ NA PO NC
2042:0104  CD21          INT     21   ←——  已执行两条指令,停在这里
-T  ←——  执行 INT 21 这条指令看看
AX=0200  BX=0000  CX=0000  DX=0003  SP=FFE8  BP=0000  SI=0000  DI=0000
DS=201D  ES=201D  SS=201D  CS=E015  IP=16B4  NV UP DI PL NZ NA PO NC
E015:16B4  80FC3F        CMP     AH,3F
-G  ↖——  进入 INT 21 服务程序(CS 程序段改变了)
  ↖——  继续执行程序的其余部分
♥  ←——  第 2 号功能调用的执行结果
AX=0203  BX=0000  CX=0000  DX=0003  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=2042  ES=2042  SS=2042  CS=2042  IP=0106  NV UP EI PL NZ NA PO NC
2042:0106  CC            INT     3   ←——  停在这里
-  ↖——  又回到原来的程序段
```

本章习题

1. 请解释下列名词:

　　倾卸 (Dump)

反汇编（Unassemble）

跟踪（Trace）

中断（Interrupt）

2. 请说明 DEBUG 工具程序的用途。

3. 请说明 DEBUG 命令的格式为何？以及两种以 DEBUG 载入 MEM. EXE 并加上/C/P 参数的方法？

4. DEBUG 的“?”命令有何用途？

5. 请以 DEBUG 的 D 命令并配合 L 命令查看存储器中 B800:0000～B800:0200中的内容。

6. 请以 DEBUG 的 A 命令键入下面的汇编语言程序，以 G 命令执行，并说明如何设定 IP 寄存器以重新执行程序？

```
MOV  AH,09
MOV  DX,109
INT  21
INT  3
DB  "HELLO ASSEMBLY $"
```

7. 将上述程序的最后一行(DB "HELLO ASSEMBLY $")调至第一行，然后依序输入程序的其余部分，请说明如何以 G 命令加上程序执行地址的方式执行此程序？(请注意 MOV DX,109必须改为 MOV DX,100才能使 DX 指向正确的字符串地址)

8. 请以 DEBUG 的 H 命令计算两个十六进制数字5678h、1234h 的和与差。

9. 试问 DEBUG 的 U 命令有何功能？

10. 请以 DEBUG 的 R 命令查看寄存器的内容，并将 AX 寄存器的内含值改为0345h。

11. 请以 R 命令将标志寄存器中的溢出及进位标志设为1。

12. 请用 DEBUG 的 T 命令和 P 命令跟踪下列程序，并说明在执行到 INT 21时，两者有何差异？(当程序跟踪进 INY 21以后，由于中断服务程序很长，我们可以直接下达 G 命令让程序一直执行到结束为止)

```
MOV DL,03
MOV  AH,02
INT  21
INT  3
```

13. 请说明如何进入 DEBUG 后使用 N、L、G 命令载入并执行 DOS 中的 CHKDSK. EXE 并加上/F 的执行参数？

14. 请以 DEBUG 中的 A 命令并配合 DB 伪指令输入下列字符串，再以 R、N、W 命令将此字符串写入名为 FILEA. TXT 的文件中。

"THIS IS FILE A."

15. 请先用 DEBUG 的 A 命令将下列汇编语言程序键入电脑中，然后再以 DEBUG 的 R、N、W 命令将此程序写入名为 BELL. COM 的文件中。

```
MOV  DL,07
MOV  AH,02
INT  21
MOV  AX,4C00
INT  21
```

第四章　汇编语言的开发流程和程序结构

学习汇编语言除了要了解指令语法之外，还要熟悉程序结构以及汇编连接的方法。本章将介绍 MASM 程序的开发流程，内容包括：

□汇编语言程序开发的流程

□汇编语言的语法

□MASM 中常用的伪指令

□MASM 的程序结构

□.COM 和.EXE 文件的差异

□汇编、连接、以及执行汇编语言程序

本章可以看做是对汇编语言的一个总体了解，其中涉及了许多概念，诸如：代名、伪指令、编辑、汇编、连接等等，至于后面的章节都是在此章的基础上将指令逐一套入运用，因此本章可以说是后面几章的基础。

4.1　汇编语言的程序开发流程

汇编语言的程序开发流程，如果将编写程序前的构思略去，则整个过程包括：编辑(Edit)、汇编(Assemble)、连接(Link)、执行(Execute)、调试(Debug)等步骤，如此周而复始，直到程序完全正确为止：

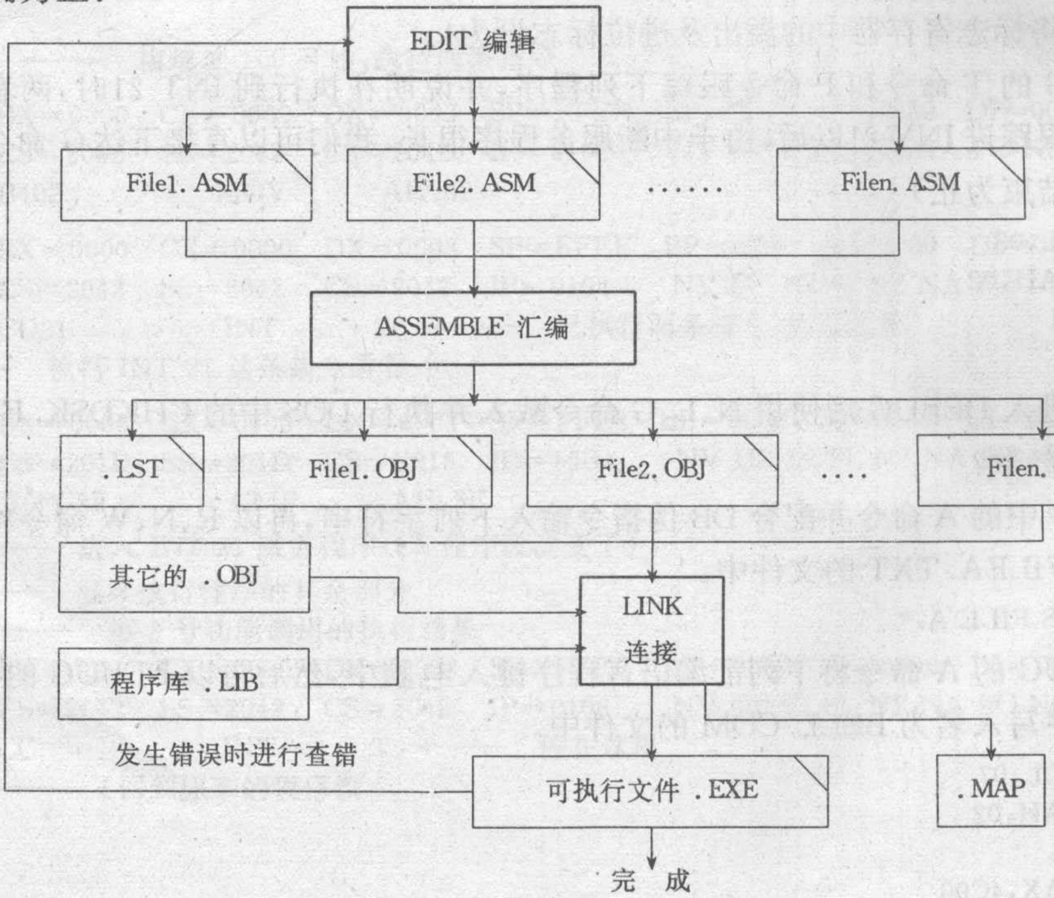


图4.1　汇编语言的开发流程

4.1.1 用编辑器来编写程序

使用 MASM 开发汇编语言程序，首先要使用编辑器(Editor)来建立原始程序，并存入磁盘中成为源文件(Source File)，其文件扩展名通常定为.ASM。

编辑器是把程序的每一个字以 ASCII 码(或是汉字内码)的形式一一存入文件中，所以我们可以用 TYPE 命令将源文件中的内容显示在屏幕上，结果与当初输入的数据完全一样。常用的编辑器除了有 EDIT(由 DOS 提供)、PE、WS、HE5等文书编辑器之外，MASM 6.x 版所附的 PWB(Program Work Bench)集成开发环境也提供了程序编辑的功能。

4.1.2 利用汇编器将源文件汇编成目的文件

由于源文件中仍忠实地存放着程序的原始面貌，因而无法直接交由 CPU 执行。汇编器(Assembler)的主要工作便是把源文件的内容转换成 CPU 所能执行的形式——机器码，而转换后的机器码文件就叫做目的文件(Object File)，其扩展名为.OBJ。

4.1.3 利用连接器将目的文件连接成可执行文件

本来目的文件是可以执行的，但是为了维持使用上的弹性，目的文件中有许多地方并未完全被翻译成可执行码。其原因有二：

1)程序可能会调用其它目的文件中的程序。由于汇编器只处理程序本身，因此对于外部调用的部分无法处理。

2)由于 PC 的存储器使用状况随时都可能会改变，因此我们无法预知程序将被载入哪一个地址上执行，所以目的文件必须先转换成可重定位的(Relocatable)机器码，才能在不同的载入地址上执行。

MASM 所提供了一个名为 LINK 的连接器，它的功能就是用来将相互调用的各个程序连接起来，并且生成可重定位的机器码文件，我们称之为可执行文件(Executable File，其扩展名为.EXE 或.COM)。

4.1.4 利用调试器来查错

较大的程序很少一次就能完全通过，所以反复调试是不可避免的。常用的调试器(Debugger)有：MS-DOS 的 DEBUG、由 MASM 6.x 所提供的 CodeView 等。DEBUG 适合初学者及小程序的调试，而 CodeView 则较适用于高级使用者及大程序。

4.2 汇编语言的语法

汇编语言程序是由一列一列的语句所组成，每条语句则是由**代名**(Name)、**操作符**(Opcode)、**操作数**(Operand)以及**注释**(Comment)四项组成的，其中除了操作符是必须的，其余则视需求可加可不加。这四项的排列顺序如下图所示，各个项之间至少空一格，至于要从哪一行开始写起，则无一定限制，我们可以利用 TAB 键将各个语句行的每个项加以对齐，以利于阅读。

代名	操作符	操作数	注释
(Name)	(Opcode)	(Operand)	(Comment)
⇩	⇩	⇩	⇩
BEGIN :	MOV	DX, OFFSET MESSAGE	; DX指向MESSAGE的段内偏移量

图4.2　汇编语言的语法格式

4.2.1　注释(Comment)

汇编语言的可读性原本就不及其它高级语言，因此当我们尝试去阅读其他人甚至自己以往编写的程序时，往往会有不知所云的感觉。所以无论是从程序的可读性角度及可维护性(Maintainable)角度来说，使用注释来说明程序内容是有必要的。

单行的注释通常以"；"开始，在汇编程序时，由"；"开始到一行结束间的数据都会被MASM视为注释而加以忽略，例如：

```
                         注  释
INT   21h           ；  这是 DOS 的中断调用
                    ↖使用 ；表示注释
```

而 Comment 这个伪指令则用于注释超过一行的情况，例如：

```
                    间隔符号
Comment  *   这是注释的第一行
         这是注释的第二行
         这是注释的第三行       *    这一部分数据也会被视为注释
```

上例中 MASM 会将 Comment 后的第一个非空白数据视为间隔符号，而在两个间隔符号之间，和最后一行的间隔符号之后的数据都会被视为注释。

4.2.2　操作符(Opcode)

操作符可分为指令和伪指令两种类型：

□指令(Instruction)：用来控制电脑做什么动作，例如：

```
MOV 操作数 1，操作数 2
↑
操作符
```

MOV 这个指令便是告诉 CPU 将第二个操作数中的数据传送到第一个操作数中。

□伪指令(Directive 或称假指令)：是用来告诉汇编器在汇编过程中要做什么，例如：

```
DB  "ABCDEFG"
↖——伪指令
```

便是要汇编器将随后的数据一个字节一个字节的逐一存入存储器中。

4.2.3 操作数(Operand)

操作数指的是操作符在运算时所需要的数据,它可能是寄存器、数字、或运算表达式等数据。有些指令不需要操作数,有些指令则需要一个、两个、或三个操作数。

```
         ↙第一操作数
MOV  AX,  5
           ↖第二操作数
```

如果在操作数中使用到了整数数据,则需在数字后面注明其数制:"b"表示二进制;"d"表示十进制;"h"则表示十六进制。MASM 的缺省值为十进制,所以当数据本身为十进制时,可以省略其数制表示符号(注意:在 DEBUG 中,缺省的数制是十六进制,和 MASM 不一样)。另外,当数据为十六进制时,如果是以 A~F 开头,则需在其前面加上一个"0"(零),汇编器才能识别出这是一个数字数据而不是一个代名。例如:

MOV AX,B000h　必须写成　MOV AX,0B000h

4.2.4 代名(Name)

在 MASM 中,我们常以英文名称来代替一些程序中会用到的地址、数据变量以及常数,这些英文名称就称为代名(Name)。代名的使用规则如下:

□代名可由英文字母、阿拉伯数字以及四个特殊字符(?,@,_,$)所组成。

□代名的长度最多可达31个字符。

□代名的第一个字符不能是数字且应避免以"@"字符开头。

□不能使用系统保留字。MASM 6.x 的保留字包括:指令名、寄存器名以及 MASM 的伪指令名。

MASM 的代名共分三种:一是程序标号、二是数据变量名、三是常数名,下面我们来看这三种代名。

<u>程序标号</u>

程序标号就是用来表明程序地址的代名,其定义方法非常简单,只要在代名的后面加上冒号即可,如以下例子中的 P1。

```
P1:MOV AX,00
   ........
   .....
   JMP P1      ;JMP 是跳转指令,执行这一行后会跳至 P1:处去执行
```

程序标号是用来代表程序中的某个地址,并做为程序流程控制之用。以上述例子而言,P1 代表的就是 MOV AX,00这一条指令的地址,当 P1做为 JMP 指令的操作数时,程序经过汇编之后,这个操作数便会被换算成 MOV AX,00这条指令的地址,因此当程序执行了 JMP P1这条指令后,CPU 便会跳到 MOV AX,00这一条指令去执行。

<u>变量名</u>

程序标号是表明程序地址的代名;而变量名则是表示数据地址的代名,这是由于数据值是可以被改变的,故称为变量名,通常使用在数据段(Data Segment)中。

以下是定义各种数据的伪指令，及其数据容量和用途对照表：

表4.1

定义数据的伪指令	定义数据范围及用途	长度
DB，BYTE	定义一个字节或一个 0～255 间的正整数	1 字节
SBYTE	定义 －128～＋127 的有符号数	1 字节
DW，WORD	定义 0～65535 的正整数	2 字节
SWORD	定义 －32,768～＋32,767 的有符号数	2 字节
DD，DWORD	定义 0～4,294,967,295（4 Giga）的正整数	4 字节
SDWORD	定义 －2,147,483,648～＋2,147,483,647 的有符号数	4 字节
DF，FWORD	定义 6 个字节的整数，通常在 386/486 的CPU上做为指针(Pointer)变量	6 字节
DQ，QWORD	定义 8 个字节的整数，用在8087系列的数学运算处理器上。	8 字节
DT，TBYTE	定义 10 个字节的整数	10 字节

举例来说：

```
VAR1  DB  ?         ;保留 1 个字节
VAR2  DB  "STRING"  ;储存 STRING 这六个字符
VAR3  DB  100       ;储存 100 这个数字
```

VAR1～VAR3均表示变量名。这三个变量名可分成两类：一种是事先设定数据值的，我们称为已初始化(Initialized)数据，如 VAR2及 VAR3分别已被设定为“STRING”和100；第二种则是在编写程序时，我们尚不知道其内容，仅保留一份数据空间给变量名，这种数据我们称为未初始化(Uninitialized)数据，未初始化的数据要以?表示，如 VAR1。

此外，我们也可以使用 DUP(Duplicate)伪指令定义重复的数据，在伪指令前面的数值代表所要储存数据的个数，而所要储存的数值或字符数据则放置在后面的括弧里。例如：

```
VAR4  DW  20 DUP(?)    ;保留 20 个未初始化的字
VAR5  DB  10DUP("*")   ;储存 10 个 * 字符
```

<u>常数名</u>

通常我们会将一些常用的数值以文字来命名，这样可以使程序看起来更容易阅读，而常数名便是用来代表一个固定不变的数值，它的后面必须伴随着一个常数值。常数名只存在于源文件中，在汇编后，所有的常数名均会被代换成常数值。

在 MASM 中我们使用 EQU 伪指令来设定常数值，例如：

```
CR     EQU     0Dh
LF     EQU     0Ah
 :
 :
MOV    AL,CR  ←—汇编后 CR 会被替换成 0Dh
```

由于0Dh 和0Ah 这两个数值在 ASCII 码中分别代表回车(Carriage Return，把光标移到第一列)和换行(Line Feed，把光标移到下一行)，所以用 CR、LF 比用0Dh、0Ah 更容易使人了解。

因此，把常用的数值或数据用 EQU 定义成容易阅读的名称，是一个良好的编程习惯。[1]

4.3 常用的 MASM 伪指令

除了已经介绍过的 DB、DW、DUP、EQU 等伪指令以外，在本节中我们再把一些 MASM 常用的伪指令分门别类的介绍给大家：

4.3.1 设置 CPU 工作模式的伪指令

MASM 6.x 缺省的 CPU 工作模式为.8086，其意思是“告诉 MASM 以8088/86工作模式来汇编”。因此，如果只需使用8088/86的基本指令时可以省略不写，而当您需要使用386、486或 Pentium 的特有指令时，则必须分别使用.386、.486、.586等伪指令来设置 CPU 的工作模式。

4.3.2 定义存储器模式的伪指令

存储器模式是用来设置程序代码及数据段的可用存储器大小。我们先举个例子来看：

```
- a
39C6:0100   BA0040      MOV  DX,4000     } 以 4000h 为数据段的起始地址，
39C6:0103   8EDA        MOV  DS,DX       } 将 [4000:110] 的值存入 AX
39C6:0105   A11001      MOV  AX,[0110]   }
39C6:0108   BA0050      MOV  DX,5000     } 以 5000h 为数据段的起始地址，
39C6:010B   8EDA        MOV  DS,DX       } 将 [5000:110] 的值存入 BX
39C6:010D   8B1E1001    MOV  BX,[0110]   }
39C6:0111   CC          INT  3
```

由上例可看出，当我们要使用不同数据段的数据时，必须先更改 DS 的值以指向正确的数据段，然后再用段内偏移量来存取数据段中的数据。

由于80x86是以64K 为一个段，因此如果程序所使用的数据合起来不超过64K，那么就可以将之集中在同一个数据段中，这样我们就只需要设定一次 DS 的值即可(也就是让 DS 固定指向此数据段的起始地址)。然而，如果程序使用的数据超过64K，那么就必须分放在不同的数据段中，此时我们则必须经常地切换 DS 的值，以存取不同数据段中的数据。

同理，如果程序段的大小在64K 以内，则可集中在同一个程序段中，我们只要设置一次 CS 的值即可；如果程序代码超过64K，则必须使用多个程序段，那么我们就得经常切换 CS 的值，以执行不同程序段中的程序。

当程序代码或数据超过64K 时，不断地切换 CS 或 DS 是件很累人的事情。还好，MASM 可以自动为我们处理这些麻烦问题。当然，我们必须在程序一开始时就告诉 MASM 要使用哪一种存储器模式，例如：

```
.286
.MODEL SMALL   ; 告诉 MASM 要使用小型存储器模式

....
```

[1] 注意：不能用 EQU 重复定义同一个常数名。

小型存储器模式(Small Model)是用来指明只使用一个数据段及一个程序段。以下为各种常用的存储器模式：

表4.2 常用的存储器模式

存储器模式	程序段	数据段
Tiny（迷你）	共用同一个段	
Small（小型）	一个	一个
Medium（中型）	多个	一个
Compact（紧密）	一个	多个
Large（大型）	多个	多个

由于 MASM 6.x 并不提供缺省的存储器模式,因此我们必须在程序一开始即使用.Model 来定义存储器模式,而且不可省略(请特别注意!)。

一般而言,除非我们的程序很大或有特殊需求,否则我们大多使用 Small 存储器模式,以省去程序中切换 DS 或 CS 的动作,这样可生成较小的执行文件,并且执行速度也较快。

另外,在使用 Tiny 模式时,由于程序段和数据段共用一个64K 的段,所以汇编后程序可转换成.COM 文件。有关.COM 和.EXE 间的差异我们将在4.7节中说明。

4.3.3 简单段定义的伪指令

MASM 6.x 所提供的段定义共有两种:一是 MASM 5.0之前所使用的完整段定义(Full Segment Definition);二是 MASM 5.0之后所提供的简单段定义(Simplified Segment Directives)。由于完整段定义太复杂而且不易使用,因此我们通常采用简单段定义伪指令来编写程序。

以下为三种常用的段定义伪指令：

□.CODE：告诉 MASM 这是程序段的开始,我们的程序便是在此段内编写。

□.DATA：表示这是一个数据段,在此段内我们可以定义所需要的数据,且容纳数据的空间不能超过64KB。

□.STACK：用来定义堆栈段。指令后面可定义堆栈的大小,如果省略系统将缺省堆栈为1024个字节。一个程序的椎栈段只能有一个。

以下即为一个 SMALL 存储器模式的汇编语言程序：

```
.MODEL SMALL    ←—— 使用 SMALL 模式
.........
.CODE
.........        } 程序段（存放程序）
.........
.DATA
.......          } 数据段（存放数据）
.........
.STACK 2048  ←—— 设定堆栈大小为 2048 字节
```

注意：当所使用的程序段超过一个以上时（Medium 或 Large 模式），可在指令后伴随段名称以示区别。例如：

```
.........
.CODE code1
.........        } 两个程序段，要给予不同的名称
.........
.CODE code2
.........
.........
```

4.3.4 **段地址宏**

@CODE、@DATA 和@STACK 是由 MASM 所提供的段地址宏，汇编程序时 MASM 会用相应的段地址取代这些宏。例如：

```
.CODE
        MOV  AX,@DATA
        MOV  ES,AX
        ..........
.DATA
MSG1    DB   "STR1"
        ..........
```

在上例中，程序经 MASM 汇编后会将@DATA 转换为数据段（即.DATA）的起始地址，因此在执行完前两行程序后，ES 会指向数据段。

4.3.5 **段地址的伪指令**（SEG）

MASM 提供了 SEG 这个伪指令可以帮助我们算出数据或程序标号的段地址。例如：

```
        - - - - -
        .CODE CODE1
        - - - - - - - -
        MOV AX,SEG DATA1   <------- 取得 DATA1 变量名的段地址
        MOV DS,AX                   (也就是 .DATA 段的段地址
        MOV AX,SEG P1      <-----
        MOV ES,AX                \
        - - - - - - - -           \
        - - - - - -                取得 p1 变量名的段地址
        .CODE CODE2                (也就是 .DATA 段的段地址
        - - - - -
P1:     XCHG AX,DX
        - - - - - - - -
        .DATA
        DATA1   DB 1,2,3,4,5
```

上例中的第一行程序的作用与 MOV AX,@DATA 是相同的,因此在执行完上述程序后 DS 会指向 DATA 这个数据段,而 ES 则会指向 CODE2这个程序段。

4.3.6 段内偏移量的伪指令(OFFSET)

相对于 SEG,伪指令 OFFSET 则可以用来求出代名所在的段内偏移量,例如:

```
段内偏移量
  0100  |              ..........
        |              MOV DX,OFFSET MSG1
        |              .........
        |              ........
        |              .DATA
  0400  |      MSG1    DB  "STR1"
        |              ........
        |              ...........
```

在上例中,DX 指向 MSG1,其值即为0400。

4.3.7 END 伪指令

END 是用来告诉 MASM 我们的程序到此结束,END 以下的文字均可忽略。因此,我们也可以将程序注释加在 END 下面,而不必使用分号或 COMMENT。此外,END 之后可以伴随一个程序标号,做为程序开始执行的地址,如果省略此标号,则 MASM 会缺省.CODE 中的第一行程序做为起始地址。

```
            .8086
            .MODEL SMALL
            .CODE
            MOV CX,02
BEGIN:      MOV AX,@DATA   ←———— 程序起始地址
            ...........
            ...........
            ...........
            END  BEGIN     ←———— 指定程序起始地址
PROGRAM IS END  ┐
            .....   ├ 注释部分
            .....   ┘
```

在上例中，从“PROGRAM IS END”到程序的结尾，MASM 都会视为注释。而且我们在 END 的后面加上 BEGIN 标号，所以程序会从 BEGIN 的地址开始执行。

4.4 MASM 的程序结构

我们以下图来说明 MASM 的程序基本结构：①

```
            .80286          ←———— 设定可使用的 CPU 指令
            .MODEL SMALL    ←———— 设定存储器模式

            CR EQU 0Dh      ┐
            LF EQU 0Ah      │
            - - - - - -     ├ 常数定义区（定义常数）
            - - - - - -     │
            - - - -         ┘

            .CODE           ┐
BEGIN:      MOV AX,@DATA    │
            - - - - - -     ├ 程序区（存放程序）
            - - - - - -     │
            - - - - - -     ┘

            .DATA           ┐
            ADDR1 DW 1234   │
            - - - -         ├ 数据区（存放数据）
            - - - - - -     ┘

            .STACK 2048     ←———— 定义堆栈大小为 2048 字节
            END BEGIN       ←———— 程序结束，并指明程序起始地址
```

图4.3　汇编语言程序的基本构造

---

① .CODE、.DATA 及 .STACK 的次序可任意调换，可用大写或小写，如果不需要数据或堆栈段，则 .DATA 或 .STACK 段可以省略。

4.5 简单的程序范例

经过前面的说明，相信读者已经十分清楚程序的整体结构，以下我们用一些简单指令来完成第一个 MASM 程序：

程序4-1 PRG4-1. ASM

```
            .8086       ；设定可使用的 CPU 指令(并非因为您的电脑为 8086 才指定为
                        ；.8086 的，这是因为 CPU 的向上兼容性！
            .MODEL SMALL                        ；小型存储器模式
CR          EQU     0Dh                         ；常数定义
LF          EQU     0Ah
            .CODE
BEGIN：     MOV     AX,@DATA
            MOV     DS,AX                       ；DS 指向数据段
            MOV     DX,OFFSET MESSAGE           ；DX 指向 MESSAGE
            MOV     AH,09h                      ；执行 09h 号 DOS 调用
            INT     21H                         ；以显示字符串
            MOV     AH,4Ch                      ；执行 4Ch 号 DOS 调用
            INT     21H                         ；以结束程序
            .DATA                               ；以下为数据段
MESSAGE     DB      "Hello! Assembly!",CR,LF
            DB      "How are you?",CR,LF,"$"
            .STACK
            END     BEGIN                       ；程序结束
```

请用 EDIT 或其它文书编辑器输入上述程序，并以 PRG4-1. ASM 为文件名存盘。

4.6 使用 ML. EXE、LINK. EXE 来汇编和连接程序

ML. EXE 是 MASM 6. x 新提供的工具程序，除了可以汇编程序外，在汇编完成后还会自动调用连接程序(LINK. EXE)将汇编好的程序(. OBJ)连接成可执行文件。事实上，MASM 6. x 提供了两种方法来完成汇编和连接的工作：

1)直接使用 ML. EXE 进行汇编和连接。

2)将汇编和连接分开，也就是先使用 MASM. EXE 进行汇编，然后再用 LINK. EXE 进行连接。

4.6.1 使用 ML. EXE

以下是 ML. EXE 的使用格式：

ML 参数 文件名 文件名 ....(各文件名之间空一格)

ML. EXE 可以一次汇编和连接数个文件，而且参数部分也可以省略。以下我们以 ML. EXE 来汇编并连接4.5节的 PRG4-1. ASM：

```
C:\MASM611>ML PRG4-1.ASM [Enter]
Microsoft (R) Macro Assembler Version 6.11
Copyright (C) Microsoft Corp 1981-1993. All rights reserved.
```

```
Assembling: PRG4-1.ASM

Microsoft (R) Segmented Executable Linker Version 5.31.009 Jul 13 1992
Copyright (C) Microsoft Corp 1984-1992. All rights reserved.

Object Modules [.obj]: PRG4-1.obj  ←———— 生成的目的文件
Run Fiel [PRG4-1.exe]: "PRG4-1.exe"  ←———— 生成的可执行文件
List File [nul.map]: NUL
Libraries [.lib]:
Definitions File [nul.def]:

C:\MASM611>
```

以下列出 ML 常用的几个参数，读者如果需要进一步的资料，可参阅附录 B。

□/Zm：用来汇编 MASM 5.x 版的程序。由于 MASM 6.1x 版除了删除了部分 MASM 5.x 版的伪指令以外，也修改了一些运算式的写法，因此为了避免更改程序的麻烦，我们只要使用参数/Zm，就可以汇编旧版的程序。

□/Fl：指定生成 LST 文件。在 LST 文件里存放着源程序对应的机器码和存储地址，以供程序员参考。

□/Fm：指定生成 MAP 文件。MAP 文件中存放各个段的相对载入地址以及程序的起始地址，以供程序员参考。

□/?：将所有的参数说明列出。

4.6.2 使用 MASM.EXE 和 LINK.EXE

MASM.EXE 会调用 ML.EXE 来进行汇编，并利用参数来抑制 ML.EXE 自动连接程序的功能，以下为汇编及连接的实例：

1)汇编程序

```
C:\MASM611>MASM PRG4-1 [Enter]
Microsoft (R) MASM Compatibility Driver
Copyright (C) Microsoft Corp 1993. All rights reserved.

  调用 ML.EXE 进行汇编
          ↓                   ↙———— 也可以汇编 5.x 版的程序
 Invoking:  ML.EXE  /I.  /Zm  /c  PRG4-1.asm
                               ↖———— 抑制自动连接功能

Microsoft (R) Macro Assembler Version 6.11
Copyright (C) Microsoft Corp 1981-1993. All rights reserverd.

 Assembling: PRG4-1.asm
C:\MASM611>
```

2)连接程序

```
C:\MASM611>LINK PRG4-1 [Enter]
Microsoft (R) Segmented Executable Linker  Version 5.31.009 Jul 13 1992
Copyright (C) Microsoft Corp 1984-1992.  All rights reserved

Run File [PRG4-1.exe]: [Enter]
List File [nul.map]: [Enter]
Libraries [.lib]: [Enter]
```

Definitions File [nul. def]: Enter

C:\MASM611>

注 意

当执行 LINK. EXE 时如果出现以下警告信息时，这表明在 CONFIG. SYS 文件中的 EMM386. EXE 驱动程序时使用了 /NOEMS 这个参数。

"Warning running program in real mode: CPU already in protected or virtual mode"

你可以不理会这个警告信息，或参考附录 C 中的说明来修改 CONFIG. SYS。

4.6.3 执行结果

无论是使用 ML 或 MASM+LINK 完成汇编和连接，其最后执行的结果都会是相同的：

```
C:\MASM611>PRG4-1  Enter
Hello! Assembly!
How are you?
C:\MASM611>
```

4.7 COM 文件和 EXE 文件

MASM 可以开发出两种可执行文件：COM 文件和 EXE 文件。这两种文件的结构有何差异呢？

4.7.1 COM 文件

COM 文件是一种最小、最简单的执行文件，其内容就是程序本身，并不附加任何其它信息。如下图所示：

图4.4 COM 文件

由于 COM 文件的程序和数据都位于同一个段内，因此其文件大小不得超过64K。DOS 在执行. COM 文件时，会直接将之载入存储器中的某个未用的段中，并设定各段寄存器的初始值，然后将控制权交给程序的第一条指令开始执行。

4.7.2 EXE 文件

EXE 文件除了程序本身之外，还含有文件头部分(Header)，参见下图：

图4.5 EXE 文件

EXE 文件头里记载着 MS-DOS 载入此执行文件时所需要的信息，例如文件大小、程序开始执行地址、堆栈寄存器和指针寄存器的初始地址、堆栈初始值等等。在执行 EXE 文件之前，DOS 会先读取文件头的数据，然后再按照这些数据决定如何载入程序、设定各个寄存器的值、以及从何处开始执行程序。

4.7.3 编写 COM 文件

COM 文件具有载入存储器快和节省存储空间的优点，在编写 COM 文件时，必须注意下列事项：

□必须使用 Tiny 存储器模式进行汇编，也就是程序中要使用.MODEL TINY 伪指令。

□在.CODE 下面加上 ORG 100h 伪指令，以告诉 MASM 将程序的起始段内偏移量设在100h 地址上。因为 DOS 在载入 COM 文件后，会把控制权交给段内偏移量100h 的地址开始执行。(COM 文件的段地址是由 DOS 在载入时自行决定的，但段内偏移量则固定为100h)

□程序中不能使用@CODE、@DATA、@STACK 等保留字。由于 COM 文件在执行时，DOS 会将 CS、DS、SS 固定指向程序的开头，所以我们不必再做任何段地址的调整。

程序4-2即是一个符合 COM 文件格式的程序：

程序4-2 PRG4-2.ASM

```
        .8086
        .MODEL TINY                          ;最小存储器模式
LF      EQU     0Ah                          ;常数定义
CR      EQU     0Dh
        .CODE
        ORG     100h                         ;程序段由 100h 开始
BEGIN:  MOV     DX,OFFSET MESSAGE            ;DX 指向 MESSAGE
        MOV     AH,09h                       ;执行 09h DOS 功能调用
        INT     21h                          ;  在屏幕上显示字符串
        MOV     AH,4Ch                       ;执行4Ch DOS 调用以结束程序
        INT     21h
        .DATA
MESSAGE DB      "Hello COM!",CR,LF,"$"
        .STACK
        END     BEGIN                        ;程序结束
```

汇编和连接

```
C:\MASM611>ML PRG4-2.ASM
Microsoft (R) Macro Assembler Version 6.11
Copyright (C) Microsoft Corp 1981-1993.  All rights reserved.

  Assembling: PRG4-2.ASM
Microsoft (R) Segmented Executable Linker  Version 5.31.009 Jul 13 1992
Copyright (C) Microsoft Corp 1984-1992.  All rights reserved.
                                              ML 会自动加上 /t 参数
Object Modules [.obj]: PRG4-2.obj/t           (因为程序是 TINY 模式)
Run File [PRG4-0.com]: "PRG4-2.com"    ←— 汇编成 COM 文件
List File [nul.map]: NUL
Libraries [.lib]:
Definitions File [nul.def]:
C:\MASM611>
```

执行结果

```
C:\MASM611>DIR PRG4-2.COM

Volume in drive C is WIND
Volume Serial Number is 253D-17CD
Directory of C:\MASM611

PRG4-2    COM          25 03-07-95   9:13a←— 只有 25 字节
        1 file(s)        25 字节
                   267,583,488 bytes free
C:\MASM611>PRG4-2
Hello COM!
C:\MASM611>
```

本章习题

1. 请解释下列名词:
   代名 (Name)
   操作码 (Opcode)
   操作数 (Operand)
   注释 (Comment)
   指令 (Instruction)
   伪指令 (Directive)
   存储器模式 (Memory Mode)
   编辑 (Edit)
   汇编 (Assemble)
   连接 (Link)
   调试器 (Debugger)
2. 试绘图说明 MASM 程序的各个组成段。
3. 汇编语言指令的格式是怎样的?请说明其各个项的功能?

4. 在命名代名时应注意哪些事项？

5. 试说明程序标号、变量名、常数名，以及它们之间的主要差异？

6. 试定义常数名 TEN 等于常数值10。

7. 什么是伪指令？伪指令与指令有何差异？

8. 请以适当的伪指令定义以下数据？

   a. 50 个字节的 2020h

   b. 一个双字 A8000024h

   c. 字符串 "Hello"

   d. 32 个未初始化的字节

9. 试说明 MASM 的存储器模式可分为哪几种？

10. 由编写汇编语言程序开始，我们还需经过哪几个步骤才能将它变成可执行文件？同时请说明各个步骤的目的？

11. 什么是调试器？它有什么作用？

12. 请将下列汇编语言程序汇编成为可执行文件，然后加以执行。

程序4-3 PRG4-3.ASM

```
         .8086
         .MODEL SMALL
LF       EQU   0Ah
CR       EQU   0Dh
         .CODE
BEGIN:   MOV   AX,@DATA
         MOV   DS,AX
         MOV   DX,OFFSET MSG1
         MOV   AH,09h
         INT   21h
         .DATA
MSG1     DB    "What is your name? ",CR,LF," $"
         .STACK
         END   BEGIN
```

13. 请在汉字环境下编写一个程序 PRG4-4.ASM，使其可在屏幕上显示下列两行文字，并将该程序汇编连接成 EXE 文件。

   “你好吗!!!”

   “这是执行 PRG4-4.ASM 程序时的信息”

14. 请说明 COM 和 EXE 文件的差异？

15. 将 PRG4-4.ASM 编写成 COM 文件格式，并加以汇编和连接。

```
C:\MASM611>ML PRG4-2.ASM
Microsoft (R) Macro Assembler Version 6.11
Copyright (C) Microsoft Corp 1981-1993. All rights reserved.
 Assembling: PRG4-2.ASM
Microsoft (R) Segmented Executable Linker Version 5.31.009 Jul 13 1992
Copyright (C) Microsoft Corp 1984-1992. All rights reserved.

Object Modules [.obj]: PRG4-2.obj
Run File [PRG4-2.com]: "PRG4-2.com"
List File [nul.map]: NUL
Libraries [.lib]:

C:\MASM611>
```

执行结果:

```
C:\MASM611>DIR PRG4-2.COM

Volume in drive C is WIND
Volume Serial Number is [illegible]
Directory of C:\MASM611

PRG4-2   COM   [illegible]
        1 file(s) [illegible]

C:\MASM611>PRG4-2
Hello COM!

C:\MASM611>
```

[illegible]

1. 请解释下列名词:
   (1) 操作码 (Opcode)
   (2) 操作数 (Operand)
   (3) [illegible] (Comment)
   (4) [illegible]
   (5) 编辑 (Edit)
   (6) 汇编 (Assemble)
   (7) 连接 (Link)
   (8) 调试器 (Debugger)
2. 试说明 MASM 程序的各个组成[illegible]
3. 汇编语言指令的格式[illegible]

第三篇　指令介绍与练习

第五章　数据传送指令集

指令的学习虽然很枯燥乏味，但这是程序的基本元素，因此请读者耐心学习，我们会尽量以容易了解的方式来解说。当然，在看过一遍之后，可能还无法全部吸收，此时可通过编写程序来加深印像及了解。

在介绍完每一个指令后，我们会立即附上一个 DEBUG 程序范例，请读者自行跟踪，以便能有更深入的了解。

本章将开始正式介绍80x86的指令集，并以使用频率最高的数据传送指令集做为敲门砖。数据传送指令集的内容包括下列四项：

□数据传送指令（MOV、XCHG）

□堆栈存取指令（PUSH、POP、PUSHF、POPF、PUSHA、POPA）

□地址操作指令（LEA、LDS、LES）

□表操作指令（XLAT）

5.1　数据传送指令

80x86 CPU 提供了一系列数据传送（Data Transferring）的指令集，现说明如下。

5.1.1　MOV 指令

MOV 是80x86中最重要的指令，由于数据每时每刻都在存储器或寄存器之间快速的移动，因此它的使用频率也最高。为了方便说明，我们在下表的语法说明里用 r 表示寄存器；segr 表示段寄存器；m 表示存储器地址；data 表示立即数据；如果命令参数中有"r/m"表示可用 r 或 m，但二者只能选一。

MOV	把源操作数送给目的操作数	
语　法：	MOV　目的操作数，源操作数	
格　式：	MOV　r1,r2	←—— r1与r2不能同时为segr r1与r2长度须一致
	MOV　r,m MOV　m,r MOV　r,data	r≠segr

说明

MOV 的动作是将右边的源操作数拷贝（Copy）至左边的目的操作数，移动之后，源操作数中的数据不会改变，而目的操作数的内含值（数据）则变成与源目的操作数的值完全一样。以下

为 MOV 使用上的限制：

□源操作数与目的操作数的长度必须一样

MOV BL,DX ——→要改成——→MOV BX,DX

□MOV 指令不能直接实现存储器单元之间的数据传送

MOV m1,m2——→要改成——→{MOV AX,m2 / MOV m1,AX}

□只有非段寄存器才能与段寄存器进行数据传送，因此要注意下列三种情况：

1)段寄存器不可以直接向段寄存器传送数据

MOV DS,CS ——→要改成——→{MOV AX,CS / MOV DS,AX}

2)存储器单元中的数据不能直接传送至段寄存器

MOV ES,m ——→要改成——→{MOV BX,m / MOV ES,BX}

3)立即数不能直接传送至段寄存器

MOV ES,0100h ——→要改成——→{MOV CX,0100h / MOV ES,CX}

读者只要记住，当操作数中有一个是段寄存器，则另一个操作数就必须是非段寄存器才行。

实　例

```
C:\>DEBUG
- a

3C0D:0100   MOV AX,1234
3C0D:0103   MOV BX,AX
3C0D:0105   MOV ES,5678
                    ^ Error 不可将立即数直接送至段寄存器
3C0D:0105   MOV DX,5678 ←—— 必须先送到一般寄存器中
3C0D:0108   MOV ES,DX ←——— 再送到段寄存器中
3C0D:010A   MOV [1100],AX
3C0D:010D   INT 3
3C0D:010E

- R                                  AX、BX 原来的值
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3C0D  ES=3C0D  SS=3C0D  CS=3C0D  IP=0100   NV UP EI PL NZ NA PO NC
3C0D:0100   B83412          MOV     AX,1234
- g ←———— 执行看看
                                     执行后 AX、BX 的值
AX=1234  BX=1234  CX=0000  DX=5678  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3C0D  ES=3C0D  SS=3C0D  CS=3C0D  IP=010D   NV UP EI PL NZ NA PO NC
3C0D:010D   CC              INT     3
- d 1100 12 ←——————— 看看 1100 地址的内容
3C0D:0100           34   12
                    ↑    ↑
-
    AL 存入低地址，AH 存入高地址
```

在上述例子中，MOV [1100],AX 程序行中的 AX 值为1234h，在执行完这一行程序后并

倾卸(Dump)[1100]的内容,会发现1234h 是以34h、12h 这种相反的顺序存入存储器的。读者只要记住高字节(High Byte)存入存储器高地址,而低字节(Low Byte)则存入存储器低地址不会混淆不清了,下图是一个字(Word)存入存储器的方法。

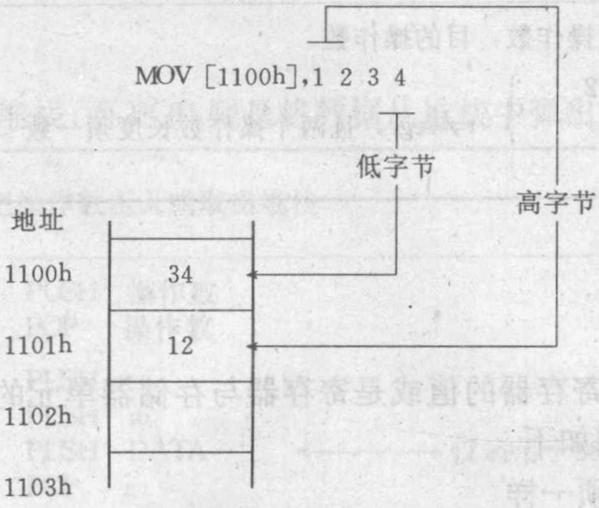


图5.1 将字数据拷贝到存储器

直接指定段寄存器(段超越)

当我们使用指针寄存器以间接寻址方式处理数据时,CPU 会预设一个段寄存器与之配合(请参阅第二章的表2.4)。例如:

```
MOV   CX,[BX]
```

由于BX 的预设段寄存器是DS,因此上述程序会将DS:[BX]的内容拷贝至CX。但是在一些特定状况下,我们可能要取得其它段的数据,此时就必须使用“段超越”(Segment Override)来设定地址的段地址,例如:

```
MOV   AX,ES:[DI]
```

则CPU 会将ES:[DI]指向的存储器内容拷贝至AX 寄存器。

指定数据类型

我们可以用“类型 PTR”来指定地址操作数的数据类型,例如:

```
MOV   AX,WORD PTR [DATA1]
MOV   WORD PTR CS:[DI],02h
........
.DATA
DATA1   DB 12,34,66,...
```

由于在数据段中我们已经定义DATA1是字节类型,因此在第一行程序中我们必须加上WORD PTR 强制指定数据为字(Word)类型,这样才能正确地将1个字(Word)的数据存入AX中,否则在汇编时会因为数据类型不匹配而产生错误。

第二行程序是把立即数02h 存入CS:[DI]的地址,如果不强制指定以字方式存入存储器,则汇编器将无法辨认CS:[DI]指向的存储器单元是要储存一个字节(Byte)还是一个字(WORD)(因为02h 可看成1个字节也可看成1个字,即0002h)。

在完成上述的两行程序后,AX 的寄存器值会等于3412,而CS:[DI]所指向的存储器单元内容则为0002h 这个字。

5.1.2 XCHG 指令

XCHG	交换两个操作数的数据
语　法：XCHG　目的操作数，目的操作数	
格　式：XCHG r1,r2 XCHG m,r XCHG r,m　　r≠segr，且两个操作数长度须一致	

说明

XCHG 指令会交换两个寄存器的值或是寄存器与存储器单元的内容，因此两个操作数都算是目的操作数，其使用限制如下：

□两个操作数的长度必须一样

XCHG　AX,BL ——→要改成——→XCHG　AX,BX

□不能交换两个存储器单元中的数据

XCHG　data1,data2——→要改成——→ { MOV　BX,data1 ; XCHG　BX,data2 ; MOV data1,BX }

□操作数不能是段寄存器

XCHG　DS,AX ——→要改成——→ { MOV　BX,DS ; XCHG　AX,BX ; MOV　DS,BX }

实　例

```
- a
3D4D:0100  MOV  AX,1234
3D4D:0103  MOV  BX,2345
3D4D:0106  XCHG  DS,AX       ←—— 段寄存器不可为操作数
                  ^ Error
3D4D:0106  XCHG  AX,BX
3D4D:0108  MOV  CL,66
3D4D:010A  XCHG  AX,CL        ←—— 两操作数长度须一致
                    ^ Error
3D4D:010A  XCHG  AL,CL
3D4D:010C  INT  3
3D4D:010D
- r                           —— AX、BX原来的值
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3D4D  ES=3D4D  SS=3D4D  CS=3D4D  IP=0100  NV UP EI PL NZ NA PO NC
3D4D:0100  B83412     MOV  AX,1234
- g  ←—— 执行看看
                                   执行后AX、BX、CX的值
AX=2366  BX=1234  CX=0045  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3D4D  ES=3D4D  SS=3D4D  CS=3D4D  IP=010C  NV UP EI PL NZ NA PO NC
3D4D:010C  CC         INT  3
```

5.2 堆栈存取指令

5.2.1 PUSH、POP 指令

PUSH 是将数据压入堆栈；而 POP 则是将数据从堆栈中弹出来。其指令格式如下：

PUSH、POP	把操作数压入或取出堆栈
语　　法：	PUSH　操作数 POP　操作数
格　　式：	PUSH　r PUSH　m PUSH　DATA　←—— 仅适用于286以上 POP　r POP　m

说明

PUSH 和 POP 的操作数可以是存储器或是寄存器。在286以上机种中，PUSH 指令则支持立即寻址方式（直接 PUSH 数据）。PUSH 及 POP 的操作数长度必须是16位。

PUSH 指令会先将 SP 指针指向上一个地址（低地址，SP 减2），然后再将操作数的值存入 SS:SP 所指向的"堆栈顶端"；而 POP 指令则会把 SS:SP 所指向的堆栈顶端的数据存入操作数中，然后再将 SP 指向下一地址（高地址，SP 加2），其示意图如下：

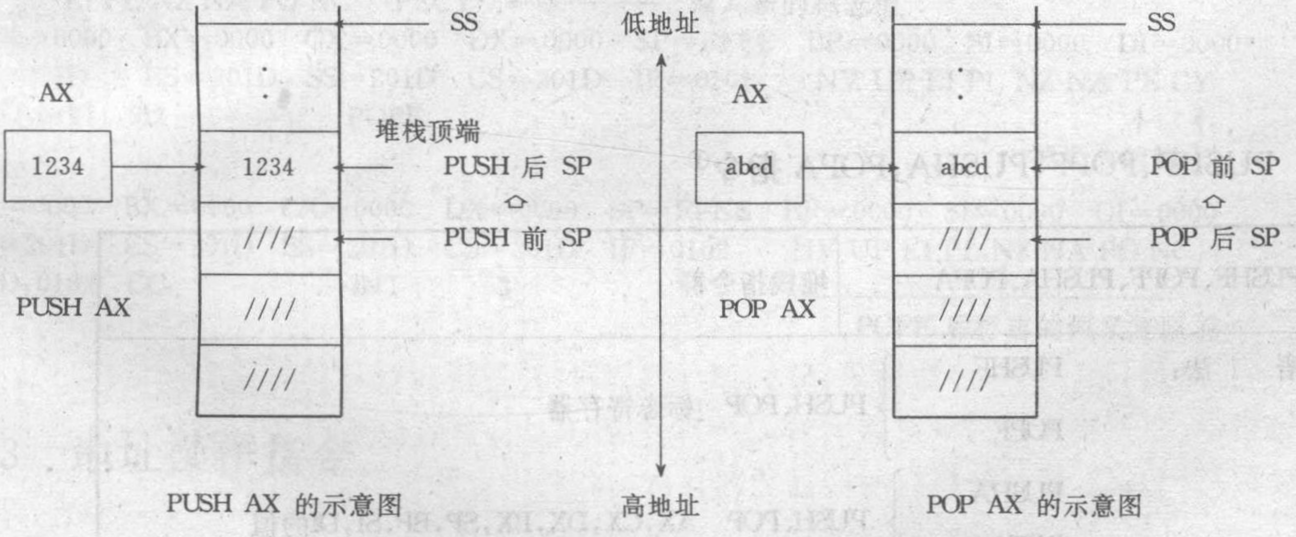


图5.2　PUSH 与 POP 的示意图.

实 例

```
- r  ←———— 程序尚未执行时，各个寄存器的初始值
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=201D  ES=201D  SS=201D  CS=201D  IP=0100   NV UP EI PL NZ NA PO NC
201D:0100  0F         DB    0F
- a
201D:0100  MOV   AX,1234
201D:0103  PUSH   AL
                 ^ Error   ←————PUSH指令操作数必须是16位
201D:0103  PUSH  AX
201D:0104  POP   BX
201D:0105  INT    3
201D:0106
- t=100 2  ←————由CS:100开始追踪2个命令
AX=1234  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=201D  ES=201D  SS=201D  CS=201D  IP=0103   NV UP EI PL NZ NA PO NC
201D:0103  50        PUSH     AX           ←——— 开始PUSH
                                                 PUSH后SP的值减2
AX=1234  BX=0000  CX=0000  DX=0000  SP=FFEC  BP=0000  SI=0000  DI=0000
DS=201D  ES=201D  SS=201D  CS=201D  IP=0104   NV UP EI PL NZ NA PO NC
201D:0104  5B        POP  BX
- dSS:FFEC ←———— 显示堆栈顶端的值
                  果然AX已被PUSH进去
201D:FFE0                                34  12                        4.
                                                 POP后SP的值加2
- t
AX=1234  BX=1234  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=201D  ES=201D  SS=201D  CS=201D  IP=0105   NV UP EI PL NZ NA PO NC
201D:0105  CC        INT       3
-
```

5.2.2 PUSHF、POPF、PUSHA、POPA 指令[①]

<table>
<tr><td>PUSHF、POPF、PUSHA、POPA</td><td>堆栈指令群</td></tr>
<tr><td colspan="2">语　法：　PUSHF / POPF　} PUSH、POP　标志寄存器
PUSHA / POPA　} PUSH、POP　AX,CX,DX,BX,SP,BP,SI,DI的值</td></tr>
</table>

① 注：PUSHA、POPA 是 80286 以上才有的指令，8088/86 无此指令

说明

这些堆栈指令后面不需接任何操作数，各个指令意义分述如下：

□PUSHF：把标志寄存器的值压入堆栈中。

□POPF：把堆栈顶端的数据弹出到标志寄存器内。

□PUSHA：是 PUSH ALL 的意思。意即按着 AX、CX、DX、BX、SP、BP、SI、DI 的顺序把这些寄存器的值依次压入堆栈中。

□POPA：是 POP ALL 的意思。与 PUSHA 的动作相反，把寄存器组按照 DI、SI、BP、SP、BX、DX、CX、AX 的次序从堆栈中弹出来。

实　例

```
-a
201D:0100   PUSHF
201D:0101   POPF
201D:0102   INT   3
201D:0103
-r  ←————————  检查执行前寄存器内容
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=201D  ES=201D  SS=201D  CS=201D  IP=0100   NV UP EI PL NZ NA PO NC
201D:0100   9C              PUSHF
-t  ←————————  单步跟踪看看
                                            ↙PUSH 后 SP 值减 2
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEC  BP=0000  SI=0000  DI=0000
DS=201D  ES=201D  SS=201D  CS=201D  IP=0101   NV UP EI PL NZ NA PO NC
201D:0101   9D              POPF
-rf  ←————————  更改标志内容
NV UP EI PL NZ NA PO NC  -PECY  ←————————  输入新的标志值
-rAX=0000  BX=0000  CX=0000  DX=0000  SP=FFEC  BP=0000  SI=0000  DI=0000
DS=201D  ES=201D  SS=201D  CS=201D  IP=0101   NV UP EI PL NZ NA PE CY
201D:0101   9D              POPF                               ↑   ↑
-t                                                          被更改的标志
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=201D  ES=201D  SS=201D  CS=201D  IP=0102   NV UP EI PL NZ NA PO NC
201D:0102   CC              INT       3                  ︸
-                                                  POPF 后标志的值又复原了
```

5.3　地址操作指令

本节所要介绍的是有关获得段地址及段内偏移量的三个指令：LEA、LDS、LES。其指令格式如下：

LEA、LDS、LES	取地址至寄存器

语　法：　LEA　r，m　⟵ 将m所指存贮单元的偏移量存入r中

LDS　r，m　⟵ 将m所指存贮单元的值，依高、低地址存入DS、r中

LES　r，m　⟵ 将m所指存贮单元的值，依高、低地址存入ES、r中

♦注：　以上的r≠Segr

说明

LEA 是将 m 的段内偏移量存至 r，而 LDS 及 LES 则是将 m 所指向的存储器内容视为“段地址:段内偏移量”，并将这一组地址存入一对寄存器之中。

5.3.1 LEA(Load Effective Address)

把存储器的段内偏移量放到指定的寄存器中，例如：

```
LEA BX,VER
```

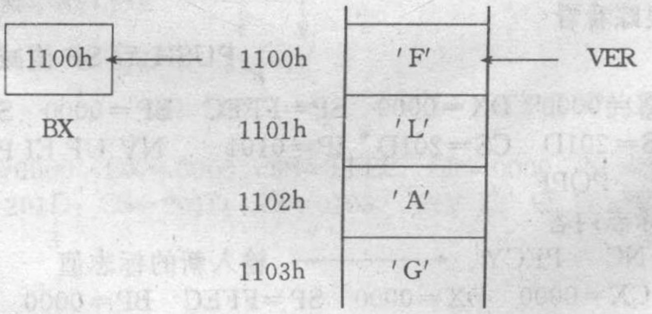


图5.3　LEA 指令动作说明

大家或许注意到了，事实上 LEA 与 OFFSET 都是用来取得数据的段内偏移量，因此以下两行指令执行的结果是相同的：

```
MOV  BX,OFFSET VAR
LEA  BX,VAR
```

所不同的是 OFFSET 为伪指令，MASM 在汇编时就把段内偏移量计算好了，而 LEA 为80x86的指令，要在程序执行时才会计算段内偏移量，因此，使用 OFFSET 时的执行速度会比较快。然而，OFFSET 却无法执行间接寻址的计算，例如：

```
        MOV  AX,OFFSET STR[BX]  ⟵ 错误用法
要改成  LEA AX,STR[BX]
```

因为在汇编时，MASM 尚不知道 BX 的值为何，必须等程序真正执行完才知道，所以只能用 LEA 指令了。

5.3.2 LDS(Load Data Segment Address)与 LES(Load Extra Segment Address)

LDS(LES)会把存储器中的双字数据中高地址的2个字节放入段寄存器DS(ES)中，而把低地址的2个字节放入指定的寄存器中。例如：

```
LDS  SI,VAR   ；表示将 5678 存入 SI 寄存器，1234 存入 DS 中
LES  BX,VAR   ；表示将 5678 存入 BX 寄存器，1234 存放 ES 中
.........
.DATA
VAR  DD  12345678h   ；5678h 存于低地址，1234 存于高地址
```

其示意图如下：

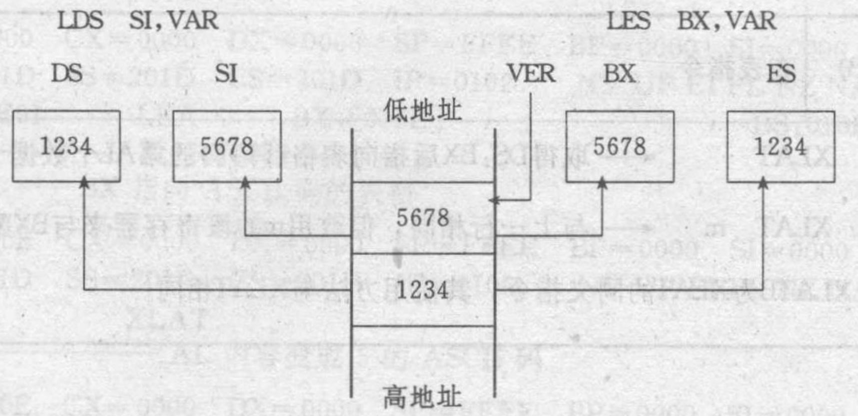


图5.4 LDS、LES 指令动作说明

实 例

```
C:\>debug
-r  ←———— 寄存器最初的状态
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3D4D  ES=3D4D  SS=3D4D  CS=3D4D  IP=0100   NV UP EI PL NZ NA PO NC
3D4D:0100   0F            DB     0F
-a  ←———— 进入汇编状态
3D4D:0100   LEA AX,[010D]
3D4D:0104   LES CX,[010D]
3D4D:0108   LDS BX,[010D]
3D4D:010C   INT 3
3D4D:010D   DW   1234←———— 低字
3D4D:010F   DW   5678←———— 高字
3D4D:0111
-t  ←———— 跟踪执行
  ↙——经过 LEA 后，AX 寄存器的值
AX=010D  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3D4D  ES=3D4D  SS=3D4D  CS=3D4D  IP=0104   NV UP EI PL NZ NA PO NC
3D4D:0104   C40E0D01      LES     CX,[010D]                DS:010D=1234
                         目前 010D 地址的内容为 1234——↗
-t
                ↙——LES 后，低的字存入 CX 寄存器中
AX=010D  BX=0000  CX=1234  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
```

```
DS=3D4D  ES=5678   SS=3D4D  CS=3D4D  IP=0108        NV UP EI PL NZ NA PO NC

3D4D:0108   C51E0D01         LDS        BX,[010D]                         DS:010D=1234
-t        LES 后,高的字存入 ES 段寄存器中
                    LDS 后,低的字存入 BX 寄存器中
AX=010D   BX=1234   CX=1234   DX=0000   SP=FFEE   BP=0000   SI=0000   DI=0000
DS=5678   ES=5678   SS=3D4D   CS=3D4D   IP=010C        NV UP EI PL NZ NA PO NC
  ——LDS 后,高的字存入 DS 段寄存器中
3D4D:010C  CC               INT        3 .
```

5.4 表操作指令

XLAT (XLATB)	查表指令
语　法：	XLAT ←— 取得DS:BX后指向表格(阵列)的第AL个数据 XLAT m ←— 与上一行相同，但改用m的段寄存器来与BX配合 ◆注： XLATB为XLAT的同义指令，其使用方法与XLAT相同

说明

XLAT(Translate by Table Lookup)的主要功能是将 DS:BX 所指向的表中的第[AL]字节中的数值返回 AL 中,例如:

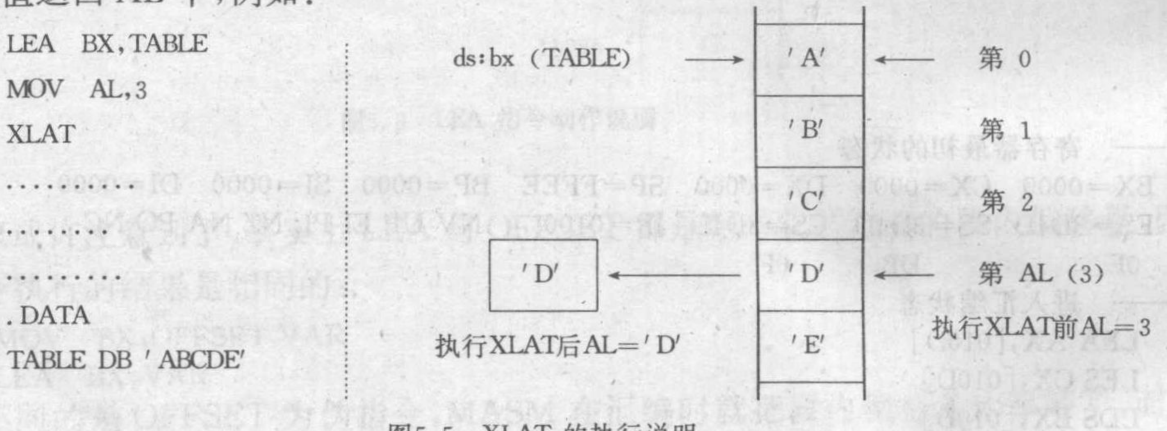


图5.5　XLAT 的执行说明

XLAT 以 DS 为缺省段地址,如果要使用别的段寄存器,则可以用"段超越"方式来指示 XLAT 到指定的段中取得数据。例如:

```
XLAT  ES:[BX]
```

则 AL 会到 ES:[BX]指向的数据表中取得数据。注意,XLAT 会固定以 BX 为段内偏移量,所以即使我们将 BX 换成其它寄存器也没用。

实　例

在以下例子中,我们用 XLAT 指令将 AL 中的数字转换为对应的 ASCII 码,然后利用

DOS 的02h 号中断调用将这个数字显示在屏幕上。

```
-a
201D:0100  MOV AL,05              ;将数值 5 存入 AL
201D:0102  LEA BX,[10E]           ;将 BX 指向数字的 ASCII 码
201D:0106  XLAT                   ;将数值 5 转换为字符'5'
201D:0107  MOV DL,AL              ;储存要显示的 ASCII 码
201D:0109  MOV AH,02              ;选择 2 号中断服务
201D:010B  INT 21                 ;执行 DOS 中断
201D:010D  INT 3
201D:010E  DB '0123456789'        ;储存数字的 ASCII 码
201D:0118
-t
          ←—————— AL 变成 5
AX=0005  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=201D  ES=201D  SS=201D  CS=201D  IP=0102   NV UP EI PL NZ NA PO NC
201D:0102  8D1E0E01      LEA     BX,[010E]                 DS:010E=3130
-t2  ←—————— 执行两条指令
          ←—————— BX 指向 ASCII 码的表格
AX=0005  BX=010E  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=201D  ES=201D  SS=201D  CS=201D  IP=0106   NV UP EI PL NZ NA PO NC
201D:0106  D7            XLAT
          ←—————— AL 内容变成 5 的 ASCII 码
AX=0035  BX=010E  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=201D  ES=201D  SS=201D  CS=201D  IP=0107   NV UP EI PL NZ NA PO NC
201D:0107  88C2          MOV     DL,AL
-g  ←—————— 执行其余部分

5  ←—————— 显示出'5'
AX=0235  BX=010E  CX=0000  DX=0035  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=201D  ES=201D  SS=201D  CS=201D  IP=010D   NV UP EI PL NZ NA PO NC
201D:010D  CC            INT     3
-
```

5.5 程序举例

用 DEBUG 学会80x86的数据传送指令后，我们进一步用这些指令编制一个 MASM 程序。

5.5.1 PRG5-1. ASM

以下程序会将存储器 FFFF:0005(BIOS 的版本日期)处的8个字符显示在屏幕上。

```
        .8086
        .MODEL SMALL
        .DATA
ADDR1   DD      0FFFF0005h      ;定义一个完整的地址 FFFF:0005h
STRING  DW      4 DUP(?)        ;保留 8 个字节储存 BIOS 日期
ENDSTR  DB      '$'             ;定义字符串结束符号
        .CODE
BEGIN:
        MOV     AX,@DATA        ;将 DATA 段地址存入 AX
        MOV     DS,AX           ;将 DS 指向 DATA 段
        LES     DI,ADDR1        ;将 ES 指向 FFFFh 段
                                ;将 0005h 存入 DI
```

```
MOV   AX,ES:[DI]     ;  ┐
MOV   STRING,AX      ;  │
MOV   AX,ES:[DI+2]   ;  │
MOV   STRING+2,AX    ;  │ 将 FFFF:0005h～FFFF:000Ch
MOV   AX,ES:[DI+4]   ;  │ 八个字节的数据拷贝至
MOV   STRING+4,AX    ;  │ STRING 地址
MOV   AX,ES:[DI+6]   ;  │
MOV   STRING+6,AX    ;  ┘

MOV   AH,09          ;  ┐
LEA   DX,STRING      ;  │ 调用中断服务程序
INT   21h            ;  ┘ 显示字符串

MOV   AX,4C00h       ;
INT   21h            ; 结束程序
.STACK               ; 设定堆栈
END   BEGIN          ; 设定 BEGIN 为程序起点
```

请将上述例子以 PRG5-1.ASM 为名存盘后，并按照第四章的说明将它汇编并连接成可执行文件。

<u>执行结果</u>

```
C:\>PRG5-1  ←———— 执行程序
04/26/94    ←———— 显示 BIOS 的版本日期
C:\>_
```

<u>程序说明</u>

1）程序首先将 DS 指向正确的数据段，以便后续的程序能正确地取得数据。

2）由于 BIOS 版本的日期是存放在 FFFF:0005h 地址上，其长度为8个字节，因此，我们用 LES 指令将 ES:DI 指向上述的地址。

3）接着再将上述地址中的8个字节的数据传送至我们所定义的字符串地址。

4）最后调用09h 字符串显示服务功能，将日期字符串显示在屏幕上。

本章习题

1. 请解释下列名词

高字节、低字节（High Byte、Low Byte）
目的操作数（Destionation Operand）
操作符（Operator）
源操作数（Source Operand）
段超越（Segment Override）
数据型态超越（Data Type Override）

2. 请说明下列指令的功能及格式？

MOV、XCHG、PUSH、POP、PUSHA、POPA、PUSHF、POPF、LEA、LDS、LES、XLAT、

3. 请问 MOV 指令在使用时有何限制？

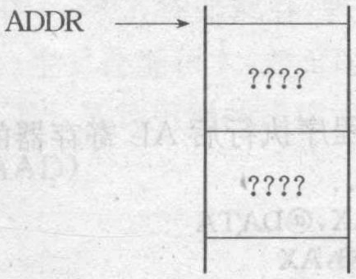
4. 请判断以下的指令是否合法，如果不合法请写出原因并加以修正。

① MOV [BX],DS
② XCHG [BX],[SI]
③ MOV [1100],1234
④ MOV SI,OFFSET MESSAGE[BX]
⑤ MOV AX,ES

5. 请问 XCHG 指令在使用时有什么限制？

6. 请问以下伪指令所定义的双字在存储器中是如何储存的？

```
ADDR  DD  ABCD1234h
```



7. 请说明在执行下列程序后各寄存器及堆栈顶端的变化情形？

执行前	AX=080Dh	BX=2CDEh	CX=0054h	SP=8234h	SS:SP=2103h

```
XCHG  AX,CX
PUSH  AX
PUSH  BX
PUSH  CX
POP   AX
PUSH  BX
POP   CX
POP   BX
```

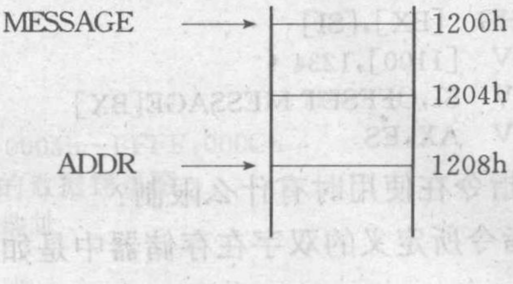
执行后	AX=	BX=	CX=	SP=	SS:SP=

8. 执行完下列的程序后，AX 及标志寄存器的值为何？

```
MOV   AX,0875h
MOV   VAR,AX
PUSH  VAR
MOV   BX,00FFh
PUSH  BX
POP   AX
POPF
```

9. 说明下列三行程序执行结束后的 BX、SI、DI、DS、ES 寄存器的值？

```
LEA BX,MESSAGE
LDS SI,ADDR
LES DI,ADDR
..........
..........
MESSAGE  DB  "HELLO"
........
ADDR  DD  102FECDAh
```



10. 试说明下列程序执行后 AL 寄存器的值？

```
..........
MOV  AX,@DATA
MOV  DS,AX
LEA  BX,TABLE
MOV  AL,04
XLAT
.........
........
.DATA
TABLE  DB  "12345678"
```

11. 用本章的指令写一个程序，将 F000:0000h～F000:000Fh 共16个字节的字符显示在屏幕上？

第六章　算术运算指令集

80x86系列CPU提供了相当完备的数学运算指令集，包含了加、减、乘、除等主要的运算。在本章中，我们将会学习到四种算术运算的基本指令以及处理BCD码的运算指令，内容包括：

□加法与减法指令(ADD、SUB)(ADC、SBB)(INC、DEC)(NEG)

□乘法指令(DIV、IDIV)

□除法指令(MUL、IMUL)

□有符号数扩展指令(CBW、CWD)

□BCD调整指令(AAA、DAA、AAS、DAS、AAM、AAD)

6.1　进位与溢出的原理

由于数学运算指令会影响到许多标志，其中最重要也最容易令人混淆的便是“进位”(Carry)与“溢出”(Overflow)标志，以下我们分别针对无符号数和有符号数、进位、溢出的意义来说明：

□无符号数与有符号数：无符号数就是将存在存储器的数值都当成正数来看待，而有符号数则是将存储器单元中的最高位视为正负号(0代表正，1代表负)，并以其它位来表示该数值的大小(参阅1.5节)。因此，同样长度的存储器空间所可以表示的有符号数及无符号数范围并不相同，见下表。

表6.1　无符号数和有符号数的表示范围

	一个字节	一个字
无符号数范围	0～255	0～65535
有符号数范围	－128～127	－32768～32767

□进位：进位标志(CF)是以无符号数的观点来判断运算是否超过可表示的范围，以两个各占1字节的数相加为例：

80h＋80h＝100h　(超过1字节可容纳的范围)

⇩　相当于10进制

128＋128＝256　(超过0～255范围)　进位(Carry)＝1

上例中两个各占1字节的数据相加后超过了一个字节可以容纳的范围，因此进位标志(CF)会被设为1。而在减法运算中，当小数减大数时，运算结果会变成负值，因为无符号数仅可表示正数，因此超出可表示范围，所以CF也会被设为1。因此，在判断是否进位时只要注意这两种状况就行了。

□溢出：溢出标志(OF)则是以有符号数的观点来判断运算结果是否超过可表示的范围，例如：

01111111b + 01111111b = 11111110b ← 两正数相加结果变为负数

符号位变为 1

⇩ 相当于 10 进制

127 + 127 = 254 (超过 1 字节有符号数可表示的范围－128～127，因此 Overflow＝1)

加法运算可能产生溢出的情况只有两种:正数加正数(可能会大于127)或负数加负数(可能小于-128)。同理,在减法时,OF＝1的情况也只有在正数减负数(相当于正数加正数)或负数减正数时(负数加负数)才可能会发生。

总之,会发生溢出的情况就是:"正加正变负"(相当于正减负变负)或"负加负变正"(相当于负减正变正)。

6.1.1 概念澄清

在第一章中我们曾经提到,电脑只认识0和1这两个数字,至于它们代表什么意思就必须靠人去解释了。虽然我们说进位和溢出是分别以无符号数和有符号数的观点来判断运算结果,但是电脑本身并没有正负数的概念,因此每一次的运算都会被电脑同时当成有符号数和无符号数的运算,以判断是否进位或溢出。例如:

超过一个字节范围

10001111b + 01111001b = 1 0001000b

⇩

8Fh + 79h = 08h, Carry＝1 ← 在无符号数运算中当成进位

在有符号数中 8Fh 会被看成 －71h

⇩ 在有符号数运算中不理会进位标志

－71h + 79h = 08h ← 结果仍在有符号数－128～127范围,溢出标志＝0

如果将这个运算当成无符号数来看,由于 Carry＝1因此其结果为108h。如果将它视为有符号数,10001111b 被当成-71h,因此运算结果变成了08h,仍在有符号数表示范围内,因此 Overflow＝0。

6.2 加法与减法指令

6.2.1 ADD、ADC 指令

ADD、ADC	加法指令
语　法：	ADD　OP1,OP2 ←—— 计算OP2＋OP1，再将结果存入OP1 ADC　OP1,OP2 ←—— 计算OP2＋OP1＋进位标志，将结果存入OP1
格　式：	ADD　r1,r2 ADD　r,m ADD　m,r ADD　r,data ADD　m,data （以上）两操作数长度须一致，r≠Segr
影响标志：	C、P、A、Z、S、O
♦ 注：	ADC的格式与ADD相同

说明

ADD(ADC)指令会把两个操作数的值相加，并把结果存入第一操作数(OP1)。在使用时有以下限制：

□OP1与 OP2的长度需一致，同为字节或是字。

□OP1可以是寄存器或是存储器地址。

□OP2可以是寄存器、存储器地址、或是立即数。

□两个操作数不能同时为存储器地址。

ADD(ADC)指令运算结果会影响 C(进位)、P(奇偶)、A(辅助进位)、Z(零值)、S(符号)、O(溢出)标志。

ADC 指令与 ADD 动作完全一样，不过 ADC 会把 C 标志的值也一并相加。所以 C 标志值为0时，ADC 的结果与 ADD 完全一样，而 C 标志为1时，ADC 的结果会比 ADD 多1。ADC 通常用于超过一个字长度的数值相加，例如31A008h＋4D7880h 时，我们可将操作数的高、低字分别放在 DX、AX 中来运算。

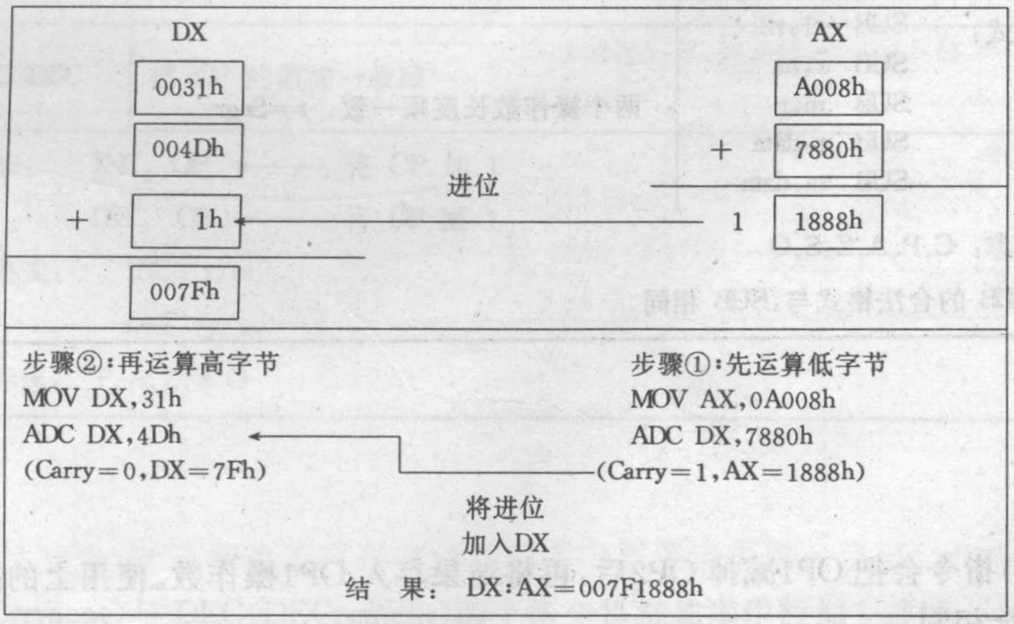


图6.1　超过字以上的加法运算

在多位数相加的场合，最低位相加要用 ADD，而接着较高位数相加则要用 ADC，因为 ADC 会把低位数相加时的进位值（C 标志值）也加进来。然而，在不考虑进位的场合，最好不要用 ADC，以免因加入 C 标志值而产生错误（因 C 值可能为1或0）。

实 例

现在我们在 DEBUG 中试试上述的例子。

```
-a
201D:0100   MOV AX,A008          ; 先运算低字
201D:0103   ADD AX,7880          ;
201D:0106   MOV DX,31            ; 再运算高字节
201D:0109   ADC DX,4D            ; 加上低字运算所产生的进位
201D:010C   INT 3
201D:010D
-g106 ←————————执行两行程序，中断点设定在 0106（低字相加）
  ↙————————运算结果
AX=1888  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=201D  ES=201D  SS=201D  CS=201D  IP=0106   NV UP EI PL NZ NA PE CY
201D:0106  B33100      MOV      DX,31      产生进位————————↗
-g ←————————执行程序剩余部分（高字节相加）
  ↙                          ↙————结果存放在 DX:AX 中（7F1888h）
AX=1888  BX=0000  CX=0000  DX=007F  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=201D  ES=201D  SS=201D  CS=201D  IP=010C   NV UP EI PL NZ NA PO NC
201D:010B  CC          INT       3
-
```

6.2.2 SUB、SBB 指令

SUB、SBB	减法指令
语　法：	SUB　OP1,OP2 ←—— 计算 OP1－OP2，结果存入OP1 SBB　OP1,OP2 ←—— 计算 OP1－OP2－进位标志，结果存入OP1
合法格式：	SUB　r1,r2 SUB　r,m SUB　m,r SUB　r,data SUB　m,data } 两个操作数长度须一致，r≠Segr
影响标志：	C、P、A、Z、S、O
＊注：SBB 的合法格式与 SUB 相同	

说明

SUB(SBB)指令会把 OP1减掉 OP2后，再将结果存入 OP1操作数。使用上的限制与 ADD(ADC)指令完全相同。

SBB(SuBstract with Borrow)则会以 OP1减去 OP2再减去 C 标志值。在无符号数的运算

中，如果以小数减大数时(OP1＜OP2)，其结果会超过无符号数的范围(负数)，则C标志会被设为1(表示发生借位)。利用这个特性，我们可以用SUB配合SBB指令来做超过一个字以上的减法：先以SUB做低字的减法，再以SBB做高字的减法即可。

实　例

以下我们用40002000h－10003000h(其值应为2FFFF000h，请自己算看看)为例，做双字的减法运算。

```
-a
201D:0100   MOV AX,2000          ；先运算低字
201D:0103   SUB AX,3000          ；
201D:0106   MOV DX,4000          ；再运算高字
201D:0109   SBB DX,1000          ；减掉低字运算所产生的借位
201D:010D   INT 3
201D:010E
-g106 ←——————执行两行程序(低字相减)
  ↙————————运算结果

AX=F000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=201D  ES=201D  SS=201D  CS=201D  IP=0106   NV UP EI NG NZ NA PE CY
201D:0106   BA0040      MOV      DX,4000          结果为负值——↗       ↑
-g ←——————执行程序剩余部分(高字相减)                               产生借位
  ↙————————————————————————结果存放在DX:AX中(2FFFF000h)

AX=F000  BX=0000  CX=0000  DX=2FFF  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=201D  ES=201D  SS=201D  CS=201D  IP=010D   NV UP EI PL NZ AC PE NC
201D:010D   CC      INT     3
-
```

6.2.3　INC与DEC指令

INC、DEC	把 OP 的值加一或减一
语　法： INC OP ←—— 将 OP 加 1 DEC OP ←—— 将 OP 减 1 合法格式： INC r/m DEC r/m 影响标志： P、A、Z、S、O	

说明

INC(INCrement)与DEC(DECrement)两个指令是将其操作数加一或减一，操作数的值可以是寄存器或存储器地址。其指令的效果等于

ADD　操作数，1

SUB 操作数,1

但 INC、DEC 不同于 ADD、SUB 的地方在于它们并不会影响 C 标志位。这两个指令常用来增减循环计数器的值,当运算后的值为0时,Z 标志会变成1,因此可以用 Z 标志来控制程序的流程。

实 例

```
-a
230F:0100   MOV AL,80
230F:0102   MOV BL,7F
230F:0104   INC BL
230F:0106   DEC AL
230F:0108   INT 3
230F:0109
-g104  ←————执行到设定 AL、BL 寄存器为止
AX=0080  BX=007F  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=230F  ES=230F  SS=230F  CS=230F  IP=0104   NV UP EI PL NZ NA PO NC
230F:0104  FEC3          INC     BL
-t
         ↙——BL=BL+1(127+1=128,超过有符号数表示范围)
AX=0080  BX=0080  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=230F  ES=230F  SS=230F  CS=230F  IP=0106   OV UP EI NG NZ AC PO NC
230F:0106  FEC8          DEC     AL            ↖——产生溢出(Overflow)
-t
         ↙——AL=AL-1(-128-1=-129,超过有符号数表示范围)
AX=007F  BX=0080  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=230F  ES=230F  SS=230F  CS=230F  IP=0108   OV UP EI PL NZ AC PO NC
230F:0108  CC            INT     3              ↖——产生溢出(Overflow)
-
```

6.2.4 NEG 指令

NEG	将 OP 的符号反相(取二进制补码)
语　法: NEG OP 合法格式: NEG r/m 影响标志: C、P、A、Z、S、O	

说明

NEG(NEGative)指令会把操作数视作有符号数并将其正负号反相;也就是说正数变负数,负数变成正数。其指令格式与前述的 INC 与 DEC 相同。

NEG 的动作是以0减掉操作数的值来得到该数的负值。话虽如此,但仍受制于有符号数所能表示范围,因此－128与－32768两个数,取负(NEG)之后却不是＋128和＋32768(超过范

围：-128～127与-32768～32767)而仍然是原来的值，但溢出标志会被设成1表示已超过了范围。

注意，只有在操作数为0时，取 NEG 之后 C 标志会设为0，其余非0操作数运算取 NEG 之后 C 标志都会设成1。

实　例

```
-a
3C0D:0100   MOV CX,8000
3C0D:0103   NEG CX
3C0D:0105   NEG DX
3C0D:0107   INT 3
3C0D:0108
-t
                                  ←——CX 的值被设成 -32768
AX=0000  BX=0000  CX=8000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3C0D  ES=3C0D  SS=3C0D  CS=3C0D  IP=0103   NV UP EI NG NZ AC PE NC
3C0D:0103  F7D9           NEG     CX
-t
                                  ←——取 NEG 之后 CX 的值并未改变
AX=0000  BX=0000  CX=8000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3C0D  ES=3C0D  SS=3C0D  CS=3C0D  IP=0105   OV UP EI NG NZ NA PE CY
3C0D:0105  F7DA           NEG     DX
-t                                        但溢出标志被设成 1,进位标志被设成 1
                                  ←——取 NEG 之后 DX 的值并未改变
AX=0000  BX=0000  CX=8000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3C0D  ES=3C0D  SS=3C0D  CS=3C0D  IP=0107   NV UP EI PL ZR NA PE NC
3C0D:0107  CC             INT     3                 但进位标志被设成 0——↗
-t
```

6.3　乘法指令

MUL、IMUL	乘法指令
语　法：	MUL　OP ←—— 无符号数 (Unsigned) 的乘法 IMUL　OP ←—— 有符号数 (Signed) 的乘法
合法格式：	MUL　r/m IMUL　r/m　} $r \neq Segr$
影响标志：	C、P、A、Z、S、O (仅 IMUL 会影响 S 标志)

6.3.1　MUL 指令

MUL(MULtiply)是无符号数相乘的指令，也许你会感到奇怪：既然是相乘，为何指令格式中只有一个操作数？另一个操作数呢？而两数相乘的结果会存在那一个寄存器中呢？答案是：

□倘若指令后面的操作数类型为字节，那么 CPU 会将该操作数乘以 AL 寄存器，然后再将结果存入 AX 寄存器。例如：

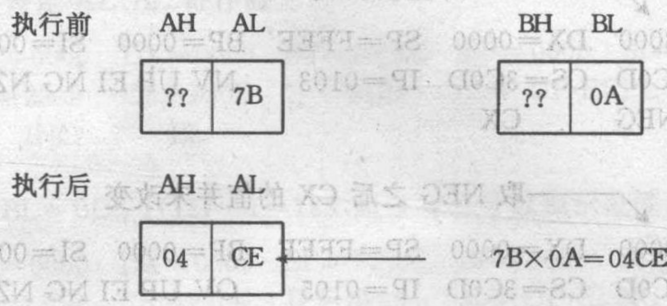
MUL BL ⇨ AX＝AL×BL

□倘若操作数的类型为字，那么 CPU 会将该操作数乘以 AX 寄存器然后再将结果存入 DX 与 AX 两个寄存器中，例如：

MUL BX ⇨ DX:AX＝AX×BX

在运算时，我们通常会将 AH 视为 AL 的延伸，将 DX:AX 视为 AX 的延伸，这是初学者必须要注意的。

由于两个 n 位乘积的长度总小于2n 个位，因此，在运算后如果产生进位及溢出，只是表示相乘的结果超过原来操作数的长度（n 个位），而不是说无法储存运算结果。相反，如果运算结果仍小于或等于操作数长度时，则 CF 与 OF 会被置为0。例如：MUL BL（假设 AL＝7B，BL＝0A）。



结果超过了一个字节，因此 Carry＝1，Overflow＝1。

实 例

```
c:\>debug
-a
3C0D:0100  MOV AL,7B
3C0D:0102  MOV BL,0A
3C0D:0104  MUL BL
3C0D:0106  MOV AX,3039
3C0D:0109  MOV CX,03E8
3C0D:010C  MUL CX
3C0D:010E  INT 3
3C0D:010F
-g106 ←———执行到 AL×BL 两个字节的相乘
      ———7B×0A 的结果存在 AX 寄存器中

AX=04CE  BX=000A  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3C0D  ES=3C0D  SS=3C0D  CS=3C0D  IP=0106   OV UP EI NG NZ AC PE CY
3C0D:0106  B83930     MOV     AX,3039
                                           结果超过 1 个字节，OF=1，CF=1

-g ←———执行到两个字的相乘
   ———————————————————结果存在 DX:AX 中
AX=5EA8  BX=000A  CX=03E8  DX=00BC  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3C0D  ES=3C0D  SS=3C0D  CS=3C0D  IP=010E   OV UP EI PL NZ NA PO CY
3C0D:010E  CC         INT     3
                                           结果超过 1 个字(Word)，OF=1，CF=1
```

6.3.2 IMUL 指令

IMUL(Integer MULtiply)是有符号数的乘法指令，与 MUL 不同的地方在于 IMUL 会将乘数与被乘数的最高位视为正负号加以处理，而不是将它视为数值的一部分。

IMUL 和 MUL 一样会将随后的操作数视为乘数，而被乘数则为 AL 或 AX，视其乘数的长度而定，运算结果则分别存放于 AX 或 DX:AX 中，读者可参考 MUL 指令的说明。

实 例

```
-a
230F:0100   MOV AL,FE          ; 将 -2 存入 AL
230F:0102   MOV BL,FE          ; 将 -2 存入 BL
230F:0104   IMUL BL            ; 将 AL×BL 存入 AX
230F:0106   MOV AX,4000        ; 将 4000h 存入 AX
230F:0109   MOV BX,FFFE        ; 将 -2 存入 BX
230F:010C   IMUL BX            ; 将 AX×BX 存入 DX:AX
230F:010E   INT 3              ;
230F:010F
-g106  ←———执行到 AL×BL 两个字节的相乘 AL×BL
       ↙———结果等于 4 (-2×-2=4)
AX=0004  BX=00FE  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=230F  ES=230F  SS=230F  CS=230F  IP=0106   NV UP EI PL NZ AC PO NC
230F:0106   B80040      MOV     AX,4000                   ↖——结果为正
-g  ←———执行到两个字的相乘 AX×BX
   ↙———————————————————————结果存在 DX:AX 中
AX=8000  BX=FFFE  CX=0000  DX=FFFF  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=230F  ES=230F  SS=230F  CS=230F  IP=010E   NV UP EI NG NZ NA PE NC
230F:010E   CC          INT     3                         ↖——结果为负
-
```

6.4 除法指令

DIV、IDIV	除法指令
语 法： DIV OP ←—— 无符号数 (Unsigned) 的除法 IDIV OP ←—— 有符号数 (Signed) 的除法 合法格式： DIV r/m, IDIV r/m } r≠Segr	

说明

和乘法指令一样，除法指令有两种：DIV 是处理无符号数的除法运算；IDIV 则是处理有符号数的除法运算。

伴随指令后的操作数代表除法中的除数，其值可以是寄存器或存储器地址，而被除数、商、及余数，则会依操作数(除数)长度的不同，而存放在不同的寄存器中：

□如果除数类型为字节，则 DIV、IDIV 会将 AX 内的值除以操作数，然后将商存于 AL 内；余数则存于 AH 之中。

□如果除数类型为一个字，则 DIV、IDIV 会将 DX:AX 的值除以操作数，然后将相除之后的商存入 AX 寄存器；余数存于 DX 寄存器。

实 例

```
-a
3C0D:0100   MOV DX,0001
3C0D:0103   MOV AX,2345
3C0D:0106   MOV BX,0100
3C0D:0109   DIV BX
3C0D:010B   INT 3
3C0D:010C
-g109  ←————执行 3 行指令
   ↙————————————————————被除数的值
                               ↙
AX=2345  BX=0100  CX=0000  DX=0001  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3C0D  ES=3C0D  SS=3C0D  CS=3C0D  IP=0109   NV UP EI PL NZ NA PO NC
3C0D:0109  F7F3          DIV      BX
-g
   ↙————商存于 AX 中          ↙——余数存于 DX 中
AX=0123  BX=0100  CX=0000  DX=0045  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3C0D  ES=3C0D  SS=3C0D  CS=3C0D  IP=010B   NV UP EI PL NZ NA PO NC
3C0D:010B  CC            INT      3
```

6.5 有符号数扩充指令

CBW、CWD	有符号数扩展指令
语　法：	CBW ←—— 将 8 位有符号数扩展为 16 位有符号数 CWD ←—— 将 16 位有符号数扩展为 32 位有符号数

说明

CBW(Convert Byte to Word)指令会将 AL 寄存器中的有符号数扩充为一个字的长度，然后存放在 AX 寄存器，而 CWD(Convert Word to Double Word)指令则会将存在 AX 寄存器里的有符号数扩充为双字长度，然后存在 DX:AX 中。

我们在1.5节正负数的表示法中曾经介绍过，在负数前可以无限制加1或在正数前加0都不会影响到原本数值的大小。符号扩充指令实际上就是用被扩充的有符号数的最高位(Signed Bit，符号位)，来填满扩充部分的位，例如：

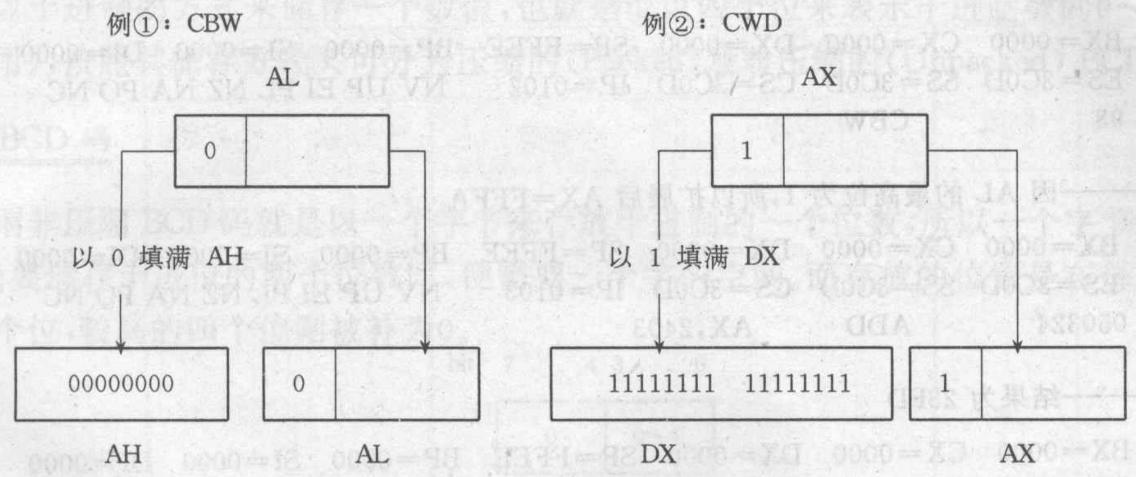


图6.2 CBW 与 CWD 动作

MASM 对操作数的长度检查很严格，如果发现操作数长度不符规定时，我们便可以先用 CBW 或 CWD 将操作数转换成较大的长度，然后再进行运算。请注意，CBW 和 CWD 指令仅能扩展有符号数，如果要扩展无符号数，则只需将扩展寄存器里填上0（正数）就可以了。下面以一个 Byte 与一个 Word 长度相加的实例来说明。

```
MOV   AL，FAh        ；将 8 位的数据（十进制的－6）存入 AL 中
CBW                  ；将 AL 的内容扩展成 16 位
ADD   AX，2403h      ；经过转换之后，就可完成字节（FAh）与
                     ；字（2403）的相加动作
```

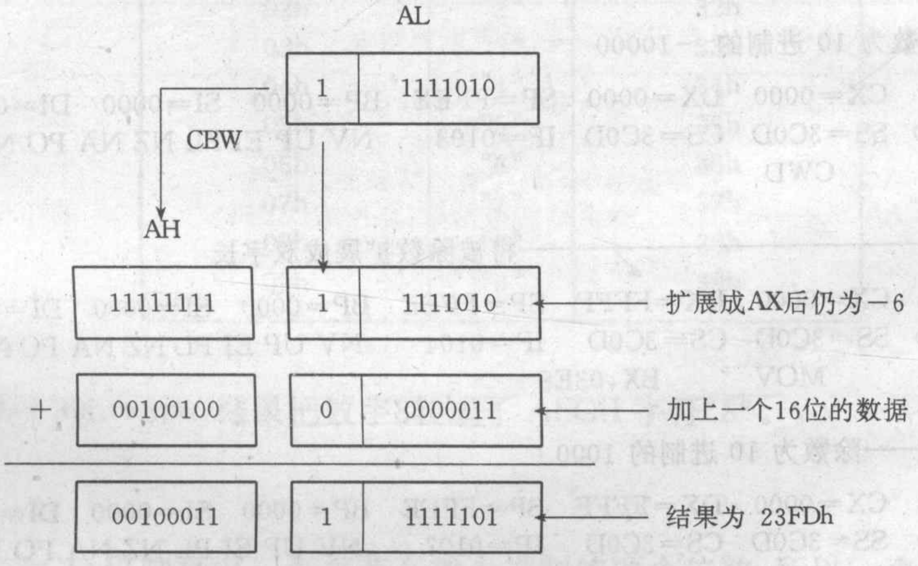


图6.3 运用 CBW 指令做不同长度的加法运算

实 例

我们将上述例子用 DEBUG 调试一下。

```
-a
3C0D:0100   MOV AL,FA
3C0D:0102   CBW
3C0D:0103   ADD AX,2403
3C0D:0106   INT 3
3C0D:0107
-t
```

```
AX=00FA  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3C0D  ES=3C0D  SS=3C0D  CS=3C0D  IP=0102   NV UP EI PL NZ NA PO NC
3C0D:0102  98            CBW
-t
```

↙——因 AL 的最高位为 1,所以扩展后 AX=FFFA

```
AX=FFFA  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3C0D  ES=3C0D  SS=3C0D  CS=3C0D  IP=0103   NV UP EI PL NZ NA PO NC
3C0D:0103  050324        ADD     AX,2403
-t
```

↙——结果为 23FD

```
AX=23FD  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3C0D  ES=3C0D  SS=3C0D  CS=3C0D  IP=0106   NV UP EI PL NZ NA PO NC
3C0D:0106  CC            INT     3
```

在介绍除法指令时,我们曾经提到当除数的长度为字时,则被除数就必须是存放在 DX:AX 的双字,因此如果遇到字与字相除的场合时,就必须运用 CWD 这个指令将被除数扩充为双字后,再做除法的运算。现在让我们来试一试以下的例子。

```
-a
3C0D:0100  MOV AX,D8F0
3C0D:0103  CWD
3C0D:0104  MOV BX,03E8
3C0D:0107  IDIV BX
3C0D:0109  INT 3
3C0D:010A
-t
```

↙——被除数为 10 进制的 -10000

```
AX=D8F0  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3C0D  ES=3C0D  SS=3C0D  CS=3C0D  IP=0103   NV UP EI PL NZ NA PO NC
3C0D:0103  99            CWD
-t
```

↙————————↙——将被除数扩展成双字长

```
AX=D8F0  BX=0000  CX=0000  DX=FFFF  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3C0D  ES=3C0D  SS=3C0D  CS=3C0D  IP=0104   NV UP EI PL NZ NA PO NC
3C0D:0104  BBE803        MOV     BX,03E8
-t
```

↙——除数为 10 进制的 1000

```
AX=D8F0  BX=03E8  CX=0000  DX=FFFF  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3C0D  ES=3C0D  SS=3C0D  CS=3C0D  IP=0107   NV UP EI PL NZ NA PO NC
3C0D:0107  F7FB          IDIV    BX
-t
```

↙——运算结果的商为 -10　　↙——余数为0

```
AX=FFF6  BX=03E8  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3C0D  ES=3C0D  SS=3C0D  CS=3C0D  IP=0109   NV UP EI PL ZR AC PE CY
3C0D:0109  CC            INT     3
```

6.6 BCD 运算调整指令

6.6.1 何谓 BCD 码

我们都知道在电脑上都是以二进制的方式来处理数值的,而 BCD(Binary Coded Decimal)

码则是以十进制的方式来储存一个数值，也就是说以四个位来表示十进制数的0～9（A～F 则舍弃不用），按照其储存方式又可分为压缩的（Packed）及非压缩的（Unpacked） BCD 码两种。

非压缩 BCD 码

所谓非压缩 BCD 码就是以一个字节来存放十进制的一个位数，所以一个字节只能存0～9的数，当要储存十进位的两个位数时，便需要一个字的空间。而存放的位置是在每个字节中较低的四个位，较高的四个位则被补为0。

bit 7 . . 4 3 . . 0

0	0～9

非压缩BCD码的 1 个位数

非压缩 BCD 码与对应数字的 ASCII 码值相差30h（数字0～9对应的 ASCII 码值为30h～31h，见下表），利用这个特性，我们只要将非压缩 BCD 码的值加上30h，便可以将它转换成对应的 ASCII 码值。

表6.2 非压缩BCD码与ASCII码对照表

非压缩BCD码	字符	字符ASCII码
00h	"0"	30h
01h	"1"	31h
02h	"2"	32h
03h	"3"	33h
04h	"4"	34h
05h	"5"	35h
06h	"6"	36h
07h	"7"	37h
08h	"8"	38h
09h	"9"	39h

例如：08h＋30h＝38h，结果把数字8转成了 ASCII 字符"8"。

压缩 BCD 码

所谓压缩 BCD 码则是以一个字节存放十进制的两个位数，所以一个字节就可存放0～99之间的某个数。至于存放方式则和非压缩 BCD 码类似，是将一个字节分成高低两个 Nibble（4个 bits），而每个 Nibble 各存放着一个十进制0～9的数。

一个字节

0～9	0～9

压缩BCD码的 2 个位数

以下为非压缩 BCD 码与压缩 BCD 码两种表示方法的对照表。

表6.3　非压缩BCD码与压缩BCD码的表示法

十进制数	非压缩BCD码		压缩BCD码	
	十位数	个位数	十位数	个位数
0		0000 0000	0000	0000
1		0000 0001	0000	0001
.		.		
.				
9		0000 1001	0000	1001
10	0000 0001	0000 0000	0001	0000
11	0000 0001	0000 0001	0001	0001
.				
.				
99	0000 1001	0000 1001	1001	1001

由上表可以清楚看出，压缩 BCD 码可以存放较多的数据，而非压缩 BCD 码则较浪费储存空间，但在转换成 ASCII 码时则较方便。

6.6.2　非压缩 BCD 码运算调整指令

AAA、AAS、AAM、AAD	非压缩BCD码运算调整指令	
语　法：AAA ←——	将非压缩BCD码相加的结果加以调整	运算结果必须放在AL
AAS ←——	将非压缩BCD码相减的结果加以调整	
AAM ←——	将非压缩BCD码相乘的结果加以调整	
AAD ←——	在除法运算前将被除数（非压缩BCD码）转换为实际的 2 进制数值	
影响标志：	A、C(AAA、AAS) S、Z、P(AAM、AAD)	

说明

由于80x86 CPU 并没有提供十进制的运算指令，因此，即使操作数是以非压缩 BCD 码形式存放，其运算仍是以二进制的方式来处理，所以我们就必须在运算后（除法则必须在运算前调整），用运算调整指令来将它调整为正确的结果。

80x86提供了四个指令，分别将加、减、乘、除的运算结果调整为非压缩 BCD 码。各个指令说明如下。

□AAA (Ascii Adjust after Addition)：是将非压缩 BCD 码的加法运算结果加以调整。例如：06＋07＝0Dh（十进制的13），经调整后 AL＝3，而 AH 会被加1，且 AF 及 CF 会被设为1。以下我们以一个 DEBUG 的实例来说明如何调整08＋07的相加结果。

```
-a
230F:0100   MOV AL,08
```

```
230F:0102   ADD AL,07
230F:0104   AAA
230F:0105   INT 3
230F:0106
-g104  ←————执行两个非压缩 BCD 码的相加 (7+8)
       ————结果为0Fh (10 进制的 15)
AX=000F  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=230F  ES=230F  SS=230F  CS=230F  IP=0104   NV UP EI PL NZ NA PE NC
230F:0104  37            AAA
-t
       ——相加结果超过10,因此 AH=AH+1,AL 被调整成5(15的个位数)
AX=0105  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=230F  ES=230F  SS=230F  CS=230F  IP=0105   NV UP EI PL NZ AC PO CY
230F:0105  CC            INT     3                            ↑       ↑
-                                       相加结果超过 10,因此 AF=1,CF=1
```

□AAS(Ascii Adjust after Substraction):将非压缩 BCD 码的减法运算结果加以调整。例如:7-8=0FFh(十进制的-1)时,经 AAS 调整后,AL=09(往上借位,相当于17-8=9),然后 AH 值会被减1,AF 及 CF 则会被设为1。以下为两位数的非压缩 BCD 码的相减(37-18)实例。

```
-a
230F:0100   MOV AL,7       ;  ⎫
230F:0102   SUB AL,8       ;  ⎬处理个位数的相减(7-8)
230F:0104   AAS            ;  ⎭
230F:0105   MOV DL,AL      ;  保存个位数相减结果
230F:0107   MOV AL,3       ;  ⎫
230F:0109   SBB AL,1       ;  ⎬处理十位数的相减(3-1-个位数的借位)
230F:010B   AAS            ;  ⎭
230F:010C   MOV DH,AL      ;  保存十位数相减结果
230F:010E   INT 3
230F:010F
-g104  ←————执行至个位数相减
       ————7-8=-1
AX=00FF  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=230F  ES=230F  SS=230F  CS=230F  IP=0104   NV UP EI NG NZ AC PE CY
230F:0104  3F            AAS
-t
       ————经调整后 AH=AH-1,AL=9(向上借位,相当于17-8=9
AX=FF09  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=230F  ES=230F  SS=230F  CS=230F  IP=0105   NV UP EI NG NZ AC PE CY
230F:0105  88C2          MOV     DL,AL                        ↑     ↑
                                                           相减产生借位
-g10B  ←————执行至十位数相减
       ————(3-1-个位数相减所产生的借位=1)
AX=FF01  BX=0000  CX=0000  DX=0009  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=230F  ES=230F  SS=230F  CS=230F  IP=010B ·  NV UP EI PL NZ NA PO NC
230F:010B  3F            AAS
-g
                          ————37-18=19(非压缩 BCD 码)
AX=FF01  BX=0000  CX=0000  DX=0109  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=230F  ES=230F  SS=230F  CS=230F  IP=010E   NV UP EI NG NZ NA PO NC
230F:010E  CC            INT     3
```

-

□AAM(Ascii Adjust after Multiplication):将乘法运算的结果调整为非压缩 BCD 码。调整方法为:先将 AL 除以10后,再将商存入 AH,余数则存入 AL。例如:

假设相乘结果 AL=51h (81)

```
51h÷10 { 商=8 ────→存入 AH }
       { 余数=1 ───→存入 AL } AX=0801h
```

我们会发现 AAM 这个调整指令实际上是将一个字节的数值转换为两位数的非压缩 BCD 码,因此在二进制数值要转换为十进制表示方式时,也经常用到这个指令,我们会在介绍 AAD 指令时一并举例说明。

□AAD(Ascii Adjust before Division):和其它三个指令不同的地方是,这个指令必须在除法运算之前使用,以便将两位数的非压缩 BCD 码(被除数)先转换为实际的二进制数值,然后再进行除法运算。其调整方法如下:

AL=AH×10+AL, AH=0

现在我们用一个二位数(37)除以一位数(3)来做例子。

```
-a
201D:0100  MOV AL,0307       ; 将被除数 37 以非压缩 BCD 码方式存入 AX
201D:0103  MOV BL,3          ; 将除数 3 存入 BL
201D:0105  AAD               ; 将 AX(非压缩 BCD 码) 先转换成 2 进制数
201D:0107  DIV BL            ; 再除以 3
201D:0109  MOV BL,AH         ; 将余数存入 BL
201D:010B  AAM               ; 再将商转换成二位数的非压缩 BCD 码
201D:010D  INT 3             ;
201D:010E
-g105  ←───执行至个位数相减
     ↙        ↙───被除数 37 存入 AX,除数 3 存入 BL
AX=0307  BX=0003  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=201D  ES=201D  SS=201D  CS=201D  IP=0105   NV UP EI PL NZ NA PO NC
201D:0105  D50A          AAD
-t
     ↙───将非压缩 BCD 码 37 转换为实际数值 25h(3×10+7=25h)
AX=0025  BX=0003  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=201D  ES=201D  SS=201D  CS=201D  IP=0107   NV UP EI PL NZ AC PO NC
201D:0107  F6F3          DIV     BL
-t
    ↙───AH 为余数,AL 为商(十进制的 12)
AX=010C  BX=0003  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=201D  ES=201D  SS=201D  CS=201D  IP=0109   NV UP EI NG NZ AC PO CY
201D:0109  88E3          MOV     BL,AH
-g
     ↙        ↙───用 AAM 将商 0Ch 转换为 0102h,余数存入 BL
AX=0102  BX=0001  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=201D  ES=201D  SS=201D  CS=201D  IP=010D   NV UP EI PL NZ NA PO NC
201D:010D  CC            INT     3
-
```

在上例中,由于经 DIV 运算后所得的商仍然是二进制数值,因此我们仍必须使用 AAM 这个指令将这个数值转换成非压缩 BCD 码。

6.6.3 压缩 BCD 码运算调整指令

DAA、DAS	压缩BCD码运算调整指令
语　法： 影响标志：	DAA ←—— 将压缩BCD码相加的结果加以调整 DAS ←—— 将压缩BCD码相减的结果加以调整 C、P、A、Z、S　　　　运算结果必须放在AL

80x86提供 DAA 和 DAS 两个指令分别用来做压缩 BCD 码加法及减法的调整指令。由于一个字节可以储存两位数的压缩 BCD 码,因此这两个指令可以一次处理两位数的运算结果,而不像 AAA 和 AAS 一次仅能处理一个位数的运算结果。以下为这两个指令的使用方法。

□DAA(Decimal Adjust al after Addition):把相加后的 AL 值调整成压缩 BCD 码。以下为37+27的运算实例。

```
-a
201D:0100   MOV AL,37
201D:0102   ADD AL,27
201D:0104   DAA     ←——转换为压缩 BCD 码的表示方式
201D:0105   INT 3
201D:0106
-g104
          ↙——37h+27h=5Eh
AX=005E  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=201D  ES=201D  SS=201D  CS=201D  IP=0104   NV UP EI PL NZ NA PO NC
201D:0104  27           DAA  ←——转换为压缩 BCD 码的表示方式
-t
          ↙——果然是 64(等于十进制 37+27)
AX=0064  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=201D  ES=201D  SS=201D  CS=201D  IP=0105   NV UP EI PL NZ AC PO NC
201D:0105  CC           INT    3              个位数产生借位——↗
-
```

□DAS(Decimal Adjust al after Substrction):把相减后的 AL 值做压缩 BCD 码调整。以下为87-28的运算实例。

```
-a
201D:0100   MOV AL,87
201D:0102   SUB AL,28
201D:0104   DAS  ←——相减结果转换为压缩 BCD 码的表示方式
201D:0105   INT 3
201D:0106
-g104
          ↙——87h-28h=5Fh
AX=005F  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=201D  ES=201D  SS=201D  CS=201D  IP=0104   OV UP EI PL NZ AC PE NC
201D:0104  2F           DAS  ←——做压缩 BCD 码的转换
-t
          ↙——果然是 59(等于十进制 87-28)
AX=0059  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
```

```
DS=201D  ES=201D  SS=201D  CS=201D  IP=0105   NV UP EI PL NZ AC PE NC
201D:0105  CC         INT     3             个位数产生借位——↗
-
```

6.7 程序举例

以下实例会将目前系统的时间显示在光标处，由于这个例子会使用到 DOS 中断的第2Ch 功能调用，以下我们先为您介绍它的使用方式。

6.7.1 功能调用2Ch 取系统时间

```
          AH 存放 2Ch
执  行：INT  21h
返回值：CH=时(0～23)
          CL=分(0～59)
          DH=秒(0～59)
          DL=秒/100(0～99)
```

解说

2Ch 功能调用会取得系统时间，并由 CX 及 DX 两个寄存器传回。

6.7.2 汇编语言程序实例

程序6-1　PRG6-1.ASM

```
        .8086
        .MODEL SMALL              ；设定为 Small 存储器模式
        .DATA
HOUR    DB    ?,?,":"             ；储存 时
MINUTEDB      ?,?,":"             ；储存 分
SECONDDB      ?,?,                ；储存 秒
ENDOFSDB      "$"                 ；储存字符串结束字符
        .CODE
BEGIN:
        MOV   AX,@DATA            ；将 DATA 段地址存入 AX
        MOV   DS,AX               ；将 DS 指向 DATA 段
        MOV   AH,2Ch              ；取系统时间
        INT   21h
        MOV   AL,CH               ；取小时
        AAM                       ；将二进制数转换成两位数的非压缩 BCD 码
        ADD   AX,3030h            ；将 BCD 码转换成相应的 ASCII 码
        MOV   HOUR,AH             ；将小时的十位数存入存储器
        MOV   HOUR+1,AL           ；将小时的个位数存入存储器
        MOV   AL,CL               ；取分钟
        AAM                       ；将二进制数转换成两位数的非压缩 BCD 码
        ADD   AX,3030h            ；将 BCD 码转换成相应的 ASCII 码
        MOV   MINUTE,AH           ；将分钟的十位数存入存储器
        MOV   MINUTE+1,AL         ；将分钟的个位数存入存储器
        MOV   AL,DH               ；取秒数
        AAM                       ；将二进制数转换成两位数的非压缩 BCD 码
```

```
ADD   AX,3030h              ；将 BCD 码转换成相应的 ASCII 码
MOV   SECOND,AH             ；将秒数的十位数存入存储器
MOV   SECOND+1,AL           ；将秒数的个位数存入存储器
MOV   AH,09                 ；DOS 09h 号服务功能
MOV   DX,OFFSET HOUR        ；DX 指向时间字符串开头
INT   21h                   ；执行调用以显示字符串
MOV   AX,4C00h              ；
INT   21h                   ；结束程序
.STACK                      ；设定堆栈
END   BEGIN                 ；设定 BEGIN 为程序起点
```

程序说明

1)程序首先执行 DOS 的2Ch 功能调用以取得系统时间,时、分、秒数分别储存于 CH、CL、DH 中。

2)依次将这三个数值转换为非压缩 BCD 码的格式,然后再转换为对应的 ASCII 码并存入字符串。

3)调用 DOS 09h 功能,将时间字符串显示在屏幕上。

本章习题

1. 请解释下列名词?

   借位 (Borrow)

   进位 (Carry)

   溢出 (Overflow)

   BCD (Binary Coded Decimal)

   压缩 BCD 码

   非压缩 BCD 码

2. 试说明以下指令的功能与格式?

   ADD、ADC、SUB、SBB、INC、DEC、NEG、DIV、IDIV、MUL、
   IMUL、CBW、CWD、AAA、AAS、AAM、AAD、DAA、DAS

3. 请指出下面程序中的错误之处并,并写出正确的程序。

```
ADD AX,BL
IMUL 20 (操作数占一个字节)
IDIV BL
```

4. 试判断下列程序执行过程中,每一步骤的 AL、Carry、及 Overflow 的值?

```
MOV AL,47h     AL =47h, C =0, O =0
ADD AL,39h       ?        ?      ?
ADD AL,F4h       ?        ?      ?
ADD AL,90h       ?        ?      ?
MOV AL,7F        ?        ?      ?
INC AL           ?        ?      ?
```

5. 试判断下列程序执行过程中,每一步骤的 AL、Carry、及 Overflow 值?

```
MOV AL,65h     AL =65h, C =0, O =0
SUB AL,80h       ?        ?      ?
SUB AL,60h       ?        ?      ?
```

```
SUB AL,A0h          ?    ?    ?
MOV AL,80h          ?    ?    ?
DEC AL              ?    ?    ?
```

6. 试判断下列程序执行过程中,每一步骤的 AX 及 DX 值?

```
MOV AX,20     AX = 20   DX = 0
MOV BX,7          ?        ?
MOV BX            ?        ?
CBW               ?        ?
CWD               ?        ?
```

7. 请将下列程序执行时,每一步骤会影响到的寄存器内容列出?

```
MOV AX,20     AX=20
DEC AX        AX=?
NEG AX        AX=?
MOV BL,2      AX=?
DIV BL        AX=?
```

8. 请将下列程序执行时,每一步骤会影响到的寄存器内容列出?

```
MOV AX,08     AX=08
ADD AL,07     AL=?
AAA           AX=?
SUB AL,3      AL=?
AAS           AX=?
```

9. 利用 AAM 写一段程序,将 Value 的值转换成字符串,然后存入 STRING 并显示在屏幕上。

```
-----
.DATA
VALUE     DB 33
STRING    DB ?,?,"$"
```

10. 利用 AAD 指令写一段程序,将 STRING 这个字符串("37")转换为实际数值(37),并将结果存入 RESULT?

```
----
.DATA
STRING    DB  "37"
RESULT    DB  ?
```

11. 请参考 PRG6-1. ASM,和以下的 DOS 显示系统日期中断功能说明,写出一个能显示系统月、日及星期几的程序?

```
          AH 存放 2Ah
执  行:   INT  21h
传回值:   CX=年 (0~2099)
          DH=月 (0~12)
          DL=日 (1~31)
          AL=星期几 (0 = 星期天 … 6 = 星期六)
```

第七章　位运算指令集

本章所要介绍的是关于位(bit)的运算。通过位运算指令我们可以对数据中的特定位做很细的调整。这些指令包括：

□逻辑运算指令(AND、OR、XOR、NOT、TEST)

□移位指令(SHR、SHL、SAR、SAL)

□循环移位指令(ROR、ROL、RCR、RCL)

7.1　逻辑运算指令

AND、OR、XOR、NOT、TEST	执行 Bit 与 Bit 之间的逻辑运算
语　　法：　AND　r/m,r/m/data OR　r/m,r/m/data XOR　r/m,r/m/data TEST r/m,r/m/data NOT　r/m	OP1 与 OP2 不可同时为存储器地址，且长度必须相同。
影响标志：　C、O、P、Z、S(其中 C 与 O 两个标志会被设为 0) NOT 指令不影响任何标志	

说明

逻辑运算指令是以位为单位来进行运算，操作数中每一个位的运算结果都不会影响其他位。

□AND 指令：会将 OP1与 OP2的对应位一一做 AND 运算后，再将结果存入 OP1。AND 在逻辑上的意义为“与”。其运算真值表如下。

表7.1　AND 运算真值表

OP1	OP2	AND 运算结果
0	0	0
0	1	0
1	0	0
1	1	1

OP1和OP2都等于 1 时，结果才为 1

```
例如：    MOV   AL,2Eh
          AND   AL,0Fh
                 ↑     ↑
                OP1   OP2
                                                8 个位分别做 AND 运算
                 ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
                 AL   0 0 1 0 1 1 1 0
       AND)     0Fh   0 0 0 0 1 1 1 1  ←——     将高 4 位设为 0
     ————————————————————————————————
                AL=   0 0 0 0 1 1 1 0  =0Eh  ←——   结果存入 OP1
```

使用 AND 运算，我们可以用 OP2上为0值的位来把 OP1对应的位清除为0，这种方法叫做位屏蔽(Bit Mask)。上例中我们就是利用这个方法将 AL 的 b4～b7四个位清除为0的。

□OR 指令：会将 OP1与 OP2的对应位一一做 OR 运算后，再将结果存入 OP1。OR 在逻辑上的意义为“或”。其运算真值表如下。

表7.2　OR 运算真值表

OP1	OP2	OR 运算结果
0	0	0
0	1	1
1	0	1
1	1	1

（后三行：OP1和OP2有一个为 1 ，结果就为 1）

使用 OR 运算，我们可以用 OP2上为1的位把 OP1上对应的位设为1：

```
例如：    MOV   AL,0E2h
          OR    AL,0Fh
                 ↑     ↑
                OP1   OP2
                                                8 个位分别做 OR 运算
                 ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
                 AL   1 1 1 0 0 0 1 0
       OR)      0Fh   0 0 0 0 1 1 1 1  ←——     将低 4 位设为 1
     ————————————————————————————————
                AL=   1 1 1 0 1 1 1 1  =EFh  ←——   结果存入 OP1
```

就是利用 OP2来将 OP1的低四位设为1。

□XOR 指令：会将 OP1与 OP2的对应位一一做 XOR 运算后，再将结果存入 OP1。XOR (Exclusive OR)在逻辑上称为“异或”。其运算真值表如下。

表7.3 XOR 运算真值表

OP1	OP2	XOR 运算结果
0	0	0
0	1	1
1	0	1
1	1	0

OP1 ≠ OP2（不同）时，结果为 1

使用 XOR 运算，我们可以用 OP2上为1的位来把 OP1对应的位反相。

```
例如： MOV  AL,0F0h
       XOR  AL,55h

                                        8 个位分别做 XOR 运算
             ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
         AL  1 1 1 1 0 0 0 0  ←——  OP1
XOR)    55h  0 1 0 1 0 1 0 1  ←——  OP2
    AL=      1 0 1 0 0 1 0 1  =A5h  ←——   结果存入 OP1
```

就是利用 OP2来将 OP1的第0、2、4、6个位反相。

□NOT 指令：只用到一个操作数，其运算方法是将操作数的每一个位反相（1变0，0变1），其真值表如下。

表7.4 NOT 运算真值表

OP1	NOT 运算结果
0	1
1	0

```
例如：  MOV  AL,0F0h
        NOT  AL

        NOT ）AL     1 1 1 1 0 0 0 0
        ————————————————————————————
        AL   =   0 0 0 0 1 1 1 1   =0Fh
```

□TEST 指令：其运算方式与 AND 指令相同，不同之处在其运算结果并不会存入 OP1，而仅影响标志值。TEST 指令通常用来对某一特定的位进行测试。例如：

```
TEST  AL,00000100b
       ↑
       └————————  测试 AL 的第 2 个位是否为 1
```

在上例中，如果 AL 第2位为1则 ZF＝0（因为 AND 运算后结果不等于0），相反，如果 AL

的第2位为0，则 ZF＝1。因此利用运算后的零值标志，我们便可以检查 OP1中特定位的值。

实 例

```
-a
3D4D:0100  MOV   AL,FF          ;使用 AL 来做逻辑运算
3D4D:0102  AND   AL,F0          ;将 AL 与 F0h 做 AND 运算
3D4D:0104  OR    AL,FF          ;将 AL 与 FFh 做 OR 运算
3D4D:0106  NOT   AX             ;将 AX 反相
3D4D:0108  TEST  AL,01          ;测试 AL 的第 0 位是不是 1
3D4D:010A  INT   3
3D4D:010B
-g104
      ↙———FF 与 F0 执行 AND 后的结果
AX=00F0  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3D4D  ES=3D4D  SS=3D4D  CS=3D4D  IP=0104   NV UP EI NG NZ NA PE NC
3D4D:0104  0CFF          OR      AL,FF
-t
      ↙———F0 与 FF 执行 OR 后的结果
AX=00FF  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3D4D  ES=3D4D  SS=3D4D  CS=3D4D  IP=0106   NV UP EI NG NZ NA PE NC
3D4D:0106  F7D0          NOT     AX
-t
     ↙———00FF 执行 NOT 后的结果
AX=FF00  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3D4D  ES=3D4D  SS=3D4D  CS=3D4D  IP=0108   NV UP EI NG NZ NA PE NC
3D4D:0108  A801          TEST    AL,01
-t
AX=FF00  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3D4D  ES=3D4D  SS=3D4D  CS=3D4D  IP=010A   NV UP EI PL ZR NA PE NC
3D4D:010A  CC            INT     3                      ↑
-                                  TEST 之后的结果为 0，所以 Z 标志被设为 1
```

7.2 移位指令

SHR、SHL、SAR、SAL	移位指令
语　法：	SHR r/m,data/CL ←—— 将 r/m 数据向右移位 data/CL 个位，并由数据的最高位向下填入 data/CL 个 0 SHL r/m,data/CL ←—— 将 r/m 数据向左移位 data/CL 个位，并由数据的最低位向上填入 data/CL 个 0 SAR r/m,data/CL ←—— 将 r/m 数据向右移位 data/CL 个位，并保留最高位 SAL r/m,data/CL ←—— 与 SHL 完全相同
影响标志：	C、P、Z、S、O、C
*注：在8086/88上data只能为1。286以上则data可以是8位数值。CL则没有次数上的限制。	

7.2.1 SHR 与 SHL 指令

SHR(SHift Right)会把 OP1中的各个位依序向右移动。其移动的次数则由 OP2的值决定，

其移动方式如下图：

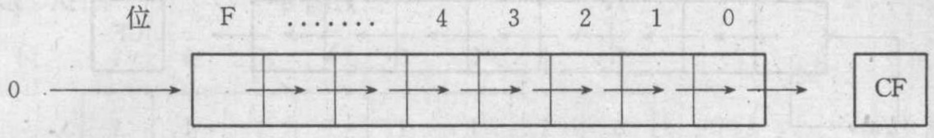


图7.1 SHR 指令移位方式

每一次右移时会先将位0的值移入 C 标志，然后依次将位1移至位0，位2移至位1……依此类推，最后将0填入最高位。

SHL(SHift Left)的使用格式与 SHR 相同，仅方向相反(向左移)而已，其移动方式如下图所示：

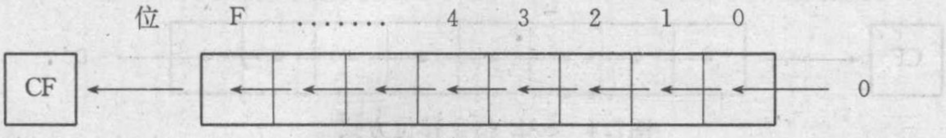


图7.2 SHL 指令移位方式

每一次左移时会先将最高位的值移入 C 标志，然后依次将次高位移至最高位……依此类推，最后将0填入最低位。

实 例

```
-a
230F:0100  MOV   AL,22
230F:0102  SHR   AL,1          ;将 AL 右移一位
230F:0104  MOV   CL,3
230F:0106  SHL   AL,CL         ;将 AL 左移三位
230F:0108  INT   3
230F:0109
-g104
           ↙———00100010b 右移一位变成 00010001b(11h)
AX=0011  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=230F  ES=230F  SS=230F  CS=230F  IP=0104   NV UP EI PL NZ AC PE NC
230F:0104  B103          MOV      CL,03
-g
           ↙———00010001b 左移三位变成 10001000b(88h)
AX=0088  BX=0000  CX=0003  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=230F  ES=230F  SS=230F  CS=230F  IP=0108   OV UP EI NG NZ AC PE NC
230F:0108  CC            INT      3
-
```

7.2.2 SAR 与 SAL 指令

SAR(Shift Arithmetic Right)与 SHR 一样是向右移位指令，只不过每次右移时最高位会保留原来的值，而不是填入0，因此并不影响数据的正负号，所以经过 n 次的 SAR 后，较高的 n 个位均会被填入最高位的值。

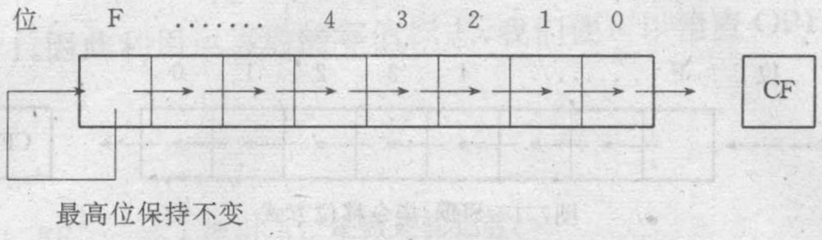


图7.3 SHR 指令移位方式

SAL(Shift Arithmetic Right)与 SHL 完全相同，它之所以存在主要是用来与 SAR 对应(注意：在 DEBUG 中并不接受 SAL，必须以 SHL 取代)。

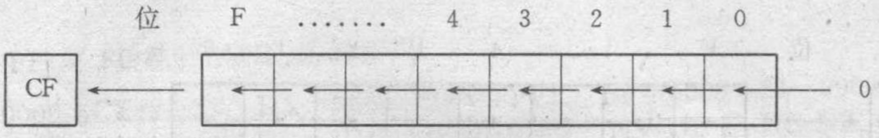


图7.4 SAL 指令移位方式

实 例

```
-a
230F:0100  MOV   AL,FE            ；将 －2 送入 AL
230F:0102  SAR   AL,1         ；将 AL 中的数做算术右移
230F:0104  MOV   CL,2
230F:0106  SAL   AL,CL            ；哦！DEBUG 不接受 SAL 指令
           ^ Error                        ⇩
230F:0106  SHL   AL,CL            ；我们改用同义指令 SHL，把 AL 中的值向左平移两位
230F:0108  INT   3
230F:0109
-g104
       ↙——11111110b 算术右移后变成 11111111b，最高位不变。
AX=00FF  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=230F  ES=230F  SS=230F  CS=230F  IP=0104   NV UP EI NG NZ AC PE NC
230F:0104  B102          MOV       CL,02
-g
       ↙——11111111b 左移两次变成 11111100b
AX=00FC  BX=0000  CX=0002  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=230F  ES=230F  SS=230F  CS=230F  IP=0108   NV UP EI NG NZ AC PE CY
230F:0108  CC            INT       3
-
```

7.2.3 移位在数学运算中的意义

位的左(右)移在数学运算中的意义相当于将该操作数乘(除)以2，就像是左(右)移一个十进制数等于将该数乘(除)以10。其中 SAR 与 SAL 由于移位时会考虑操作数的正负值，因此称为“算术移位”，可用于有符号数的乘除法。而 SHR、SHL 在位移时并不考虑正负值，因此称为“逻辑移位”，适用于无符号数的乘除法。

由于移位运算会比乘(除)法运算快得多，因此在需要乘(除)以2的次方时，我们会使用移位指令来增加程序执行效率。以下为－2÷2的例子：

```
MOV   AH,0FEh          ; FEh 即有符号数的 -2
SAR   AH,1             ; 除以 2
```

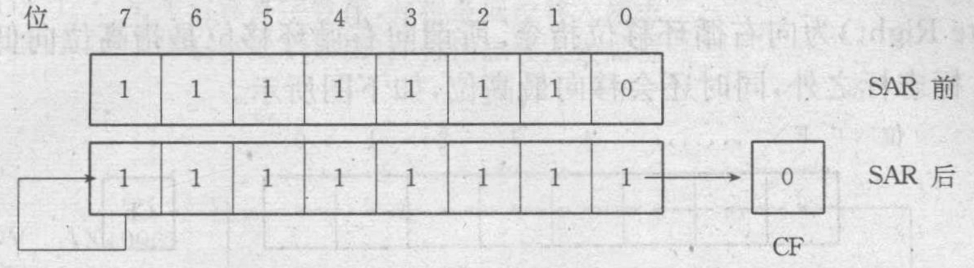


图7.5 －2 除以 2 的执行过程

结果变成11111111b。因为最高位为1表示为负数，因此取其补数变成00000001b＝1，故结果为－1。

实 例

```
-a
3D4D:0100  MOV   AL,82
3D4D:0102  SHR   AL,1
3D4D:0104  MOV   BL,82
3D4D:0106  SAR   BL,1
3D4D:0108  INT   3
3D4D:0109
-g104
       ←——以 SHR 右移,最高位补 0
AX=0041  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3D4D  ES=3D4D  SS=3D4D  CS=3D4D  IP=0104   OV UP EI PL NZ AC PE NC
3D4D:0104  B382          MOV     BL,82
-g
           ←——以 SAR 右移,最高位保持不变(即保持正负号)
AX=0041  BX=00C1  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3D4D  ES=3D4D  SS=3D4D  CS=3D4D  IP=0108   NV UP EI NG NZ AC PO NC
3D4D:0108  CC            INT     3
-
```

7.3 循环移位指令

ROR、ROL、RCR、RCL	循环移位指令
语　法：	ROR r/m,data/CL ←— 将 r/m 数据向右循环移位 data/CL 个位 ROL r/m,data/CL ←— 将 r/m 数据向左循环移位 data/CL 个位 RCR r/m,data/CL ←— 将 C 标志与 r/m 数据合在一起向右循环移位 data/CL 个位 RCL r/m,data/CL ←— 将 C 标志与 r/m 数据合在一起向左循环移位 data/CL 个位
影响标志：	C、P、Z、S、O

＊注：在8086/88上data只能为1。286以上则data可以是8位数值。CL则没有次数上的限制。

7.3.1 ROR 与 ROL 指令

ROR(ROtate Right)为向右循环移位指令。所谓向右循环移位是指高位向低位移动，而最低位除了移入 C 标志标之外，同时还会移向最高位，如下图所示。

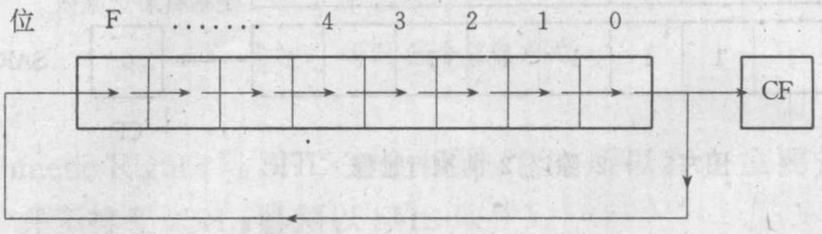


图7.6 ROR 指令循环移位方式

ROL(ROtate Left)为向左循环移位指令。其方向与 ROR 相反：

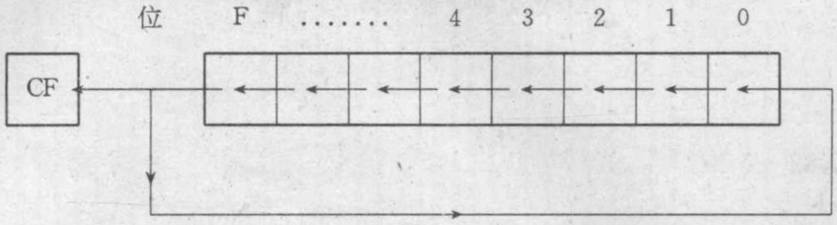


图7.7 ROL 指令循环移位方式

7.3.2 RCR 与 RCL 指令

RCR 与 RCL 也是循环移位指令，所不同的是 C 标志也加入了循环移位的行列：

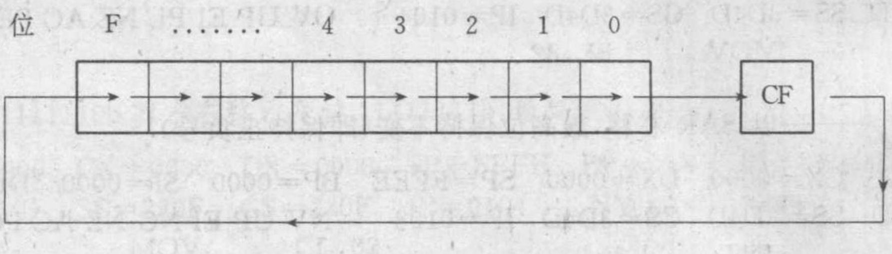


图7.8 RCR 指令循环移位方式

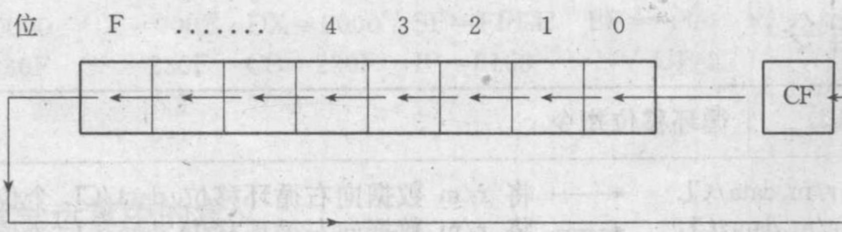


图7.9 RCL 指令循环移位方式

由于 RCL(RCR)会将 C 标志存入最低(高)位，因此可以配合 SHL(SHR)指令进行一个字以上数据的移位。例如左移 DX:AX 中的双字数据：

```
SHL   AX,1     ；先左移低字数据，并将最低位移入标志
RCL   DX,1     ；再左移高字数据，并将标志移入最低位
```

上例中左移一个双字数据时，我们必须先处理低字再处理高字；而右移时，处理次序正好

相反。例如右移 DX:AX 中的双字数据:

```
        SHR   DX,1      ;先右移高字数据,并将最低位移入标志
        RCR   AX,1      ;再右移低字数据,并将标志移入最高位
```

实 例

```
-a
61BA:0100   MOV   AX,0001
61BA:0103   ROR   AX,1
61BA:0105   RCL   DX,1
61BA:0107   INT   3
61BA:0108
-g105
       ┌───0001h 向右循环移位,最低位 1 移至最高位变成 8000h
AX=8000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=61BA  ES=61BA  SS=61BA  CS=61BA  IP=0105   OV UP EI PL NZ NA PO CY
61BA:0105  D1D2          RCL        DX,1        最低位 1 移至 CF 标志──┘
-t
                                ┌──将 CF 标志与 0000h 合并,向左循环移位变成 0001h
AX=8000  BX=0000  CX=0000  DX=0001  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=61BA  ES=61BA  SS=61BA  CS=61BA  IP=0107   NV UP EI PL NZ NA PO NC
61BA:0107  CC            INT        3           最高位 0 移入 CF 标志──┘
-
```

7.3.3 移位及循环移位时的溢出标志判断

移位或循环移位指令实际上就是乘(除)以2。然而,当我们将数据视为有符号数时,如果移位或循环移位后最高位(符号位)产生变化(正负值改变),则表示有错误发生,我们可以用溢出标志来判断这种情况(当 OF=1表示正负号发生变化)。

要特别注意的是,我们虽然可以用单一个指令来做多个位的移位或循环移位(例如:SHL AL,4),但 CPU 仍是分次来做移位,因此是否溢出,是以最后一次移位的情形为准。例如:

```
            MOV   AL,0A0h
            SHL   AL,2          ;向左平移两位
      ┌───
      相当于
      └───{ SHL   AL,1          ;第一次左移, AL 由 A0h 变成 40h
          { SHL   AL,1          ;第二次左移, AL 由 40h 变成 80h
                                 正负号改变, OF=1
```

由于最后一次位移(40h 变成80h)造成正负号改变,所以产生溢出。

7.4 程序举例

本例从键盘读取一个字符,然后将这个字符的 ASCII 值以十六进制格式显示出来。为了便于解说,下面先介绍一个读取键盘输入的 DOS 功能调用。

7.4.1 等待并读取键盘数据

AH 存放 01h

执　行:INT 21h

返回值:AL＝所读取的字符

说明

01h 调用会等待用户从键盘输入一个字符,然后将此字符显示在屏幕上,并将其 ASCII 码存入 AL 寄存器。

7.4.2 程序举例

PRG7-1. ASM　程序

```
        .286        ;使用286的指令(因程序中用到 SHR AL,4这个指令)
        .MODEL SMALL
        CR      EQU     0Dh
        LF      EQU     0Ah
        .DATA
CH-LINE DB      CR,LF
CHAR    DB      ?,?,"h $"               ;字符 ASCII 码的两个十六进制数
H-TABLE DB      "0123456789ABCDEF"      ;十六进制数值与 ASCII 码对照表
        .CODE
BEGIN:  MOV     AX,@DATA
        MOV     DS,AX                   ;将 DS 指向数据段
        MOV     AH,01                   ;输入一个字符
        INT     21h                     ;执行中断服务程序
        MOV     AH,AL                   ;将字符的 ASCII 码复制一份至 AH
        AND     AL,00001111b            ;取字符的 Low Nibble
        MOV     BX,OFFSET H-TABLE       ;将 BX 指向十六进制数值与
                                        ;ASCII 码对照表
        XLATB                           ;转换为对应的 ASCII 码
        MOV     CHAR+1,AL               ;存入存储器
        MOV     AL,AH                   ;再由 AH 取出输入的字符
        SHR     AL,4                    ;右移 4 次(将 High Nibble 移至 Low Nibble)
        XLATB                           ;转换为对应的 ASCII 码
        MOV     CHAR,AL                 ;存入存储器
        MOV     AH,09                   ;DOS 09h 号服务功能
        MOV     DX,OFFSET CH-LINE       ;DX 指向字符串开头
        INT     21h                     ;执行中断调用以显示字符串
        MOV     AX,4C00h                ;结束程序
        INT     21h
        .STACK                          ;设定堆栈
        END     BEGIN
```

执行结果

执行 PRG7-1. EXE 的结果

```
C:\>PRG7-1
A ←———输入一个字符
41h ←———在屏幕上显示此字符的 ASCII 码
C:\>_
```

说明

上述程序会将所读取字符的 ASCII 码以十六进制格式显示在屏幕光标位置上。由于一个 ASCII 码是由两个十六进制数所构成的(两个 Nibbles,各占4个位),因此我们先把读取的字符分成两个 Nibble 来处理:在处理 Low Nibble 时,我们用0Fh 和该字符做 AND 运算,如此便可将 High Nibble 清除为0,然后再用 XLATB 转换成十六进制的0~F 的字符,并存入字符串中。当要处理 High Nibble 时,则可将这个字符往右移4个位,然后再使用 XLATB 转换成0~F 的字符。

本章习题

1. 请解释以下两个名词,并指出两者的区别。

   逻辑移位 (Logic Shift)

   算术移位 (Arithmetic Shift)

2. 请说明以下指令的功能。

   AND、OR、XOR、NOT、TEST、SHR、SHL、SAR、SAL、ROR、ROL、RCR、RCL

3. 请写出下列程序执行时每一步骤的 AL 及 ZF 标志值?

```
MOV   AL,0Fh     AL =0Fh,   ZF =0
OR    AL,0A0h       ?          ?
TEST  AL,10h        ?          ?
AND   AL,34h        ?          ?
NOT   AL,           ?          ?
XOR   AL,0DBh       ?          ?
```

4. 请写出下列程序执行时每一步骤的 AL 寄存器、CF 及 OF 标志值?

```
MOV   AL,70h     AL =70h,   CF =0,   OF =0
SHL   AL,1          ?          ?         ?
SHL   AL,1          ?          ?         ?
SHR   AL,1          ?          ?         ?
```

5. 请写出下列程序执行时每一步骤的 AL 寄存器、CF 及 OF 标志值?

```
MOV   AL,7FH     AL =7Fh,   CF =0,   OF =0
SAR   AL,1          ?          ?         ?
SAL   AL,1          ?          ?         ?
SAL   AL,1          ?          ?         ?
```

6. 请写出下列程序执行时每一步骤的 AL 寄存器、CF 及 OF 标志值?

```
MOV   AL,0A7h    AL =A7h,   CF =0,   OF =0
MOV   CL,3
SAR   AL,CL         ?          ?         ?
```

```
MOV   CL,4
SAL   AL,CL          ?          ?          ?
SHR   AL,1           ?          ?          ?
```

7. 当执行下列两行程序后，AL 寄存器的值并未改变，但 CF 及 OF 却被设为1，请绘图说明这两个标志值改变的原因？

```
MOV   CL,2
MOV   AL,0AAh        AL=0AAh,   CF=0,   0F=0
ROR   AL,CL          AL=0AAH,   CF=1,   0F=1
```

8. 请写出下列程序执行时每一步骤的 AL 寄存器、CF、及 OF 标志值？

```
MOV   AL,0B2h        AL =B2h,   CF =0,   OF =0
RCR   AL,1           ?          ?          ?
MOV   AL,2
RCR   AL,CL          ?          ?          ?
MOV   CL,3
RCL   AL,CL          ?          ?          ?
```

9. 使用 SHL 及 RCL 指令写一段程序，做一个双字01112222h 乘以4的运算并将结果存在 DX：AX。

10. 改写 PRG7-1. ASM 使这个程序能读取两个字符，并将二者的 ASCII 码做 XOR 运算，然后以十六进制格式显示出这两个字符的 ASCII 码以及运算结果。

11. 改写 PRG7-1. ASM，使其能将输入字符的 ASCII 码中的第1、3、5、7位清0，然后以十六进制格式显示出来。

第八章　程序流程控制指令集

流程控制指令的作用是用来控制程序执行的走向，以便视情况来更改程序执行的顺序，或是循环(LOOP)执行某段程序。这些指令包括：

- □标志设定指令(CLC、STC、CMC、CLD、STD、CLI、STI)
- □比较指令(CMP)
- □跳转指令(JMP、Jxx)
- □循环指令(LOOP、JCXZ、LOOPxx)
- □子程序调用(CALL)
- □中断指令(INT)

8.1　标志设定指令

标志设定指令虽然没有直接用来控制程序流程，但由于多数的流程控制指令都是根据标志值来决定程序走向，所以我们在本章首先介绍标志设定指令。

8.1.1　进位标志设定指令 CLC、STC、CMC

CLC、STC、CMC	设定进位标志
语法：　CLC ←——— 将 C 标志清除为 0 　　　　STC ←——— 将 C 标志设定为 1 　　　　CMC ←——— 将 C 标志值反相(1变0，0变1) 标志：　C	

说明

80x86提供了 STC(SeT Carry flag)、CLC(CLear Carry flag)、以及 CMC(CoMplement Carry flag)这三个指令用于修改进位标志，其中 STC 指令会将 CF 设为1(DEBUG 中称为 CY)，CLC 则设定 CF 为0(在 DEBUG 中称为 NC)，而 CMC 指令则会将 CF 的值予以反相。

实　例

```
-a
39E4:0100    STC          ；将 CF 设为1
39E4:0101    CLC          ；将 CF 清 0
39E4:0102    CMC          ；将 CF 的值反相
39E4:0103
-r ←———看看标志值
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=39E4  ES=39E4  SS=39E4  CS=39E4  IP=0100    NV UP EI PL NZ NA PO NC
39E4:0100    F9           STC                       目前 CF 值为 0——↗
-t  ←———单步执行一行—↗
```

```
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=39E4  ES=39E4  SS=39E4  CS=39E4  IP=0101   NV UP EI PL NZ NA PO CY
39E4:0101   F8            CLC                    CY 表示 CF 被设定为 1——↗
-t
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=39E4  ES=39E4  SS=39E4  CS=39E4  IP=0102   NV UP EI PL NZ NA PO NC
39E4:0102   F5            CMC                    NC 表示 CF 被清除为 0——↗
-t
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=39E4  ES=39E4  SS=39E4  CS=39E4  IP=0103   NV UP EI PL NZ NA PO CY
39E4:0103   8AFF          MOV     BH,BH                  CF 反相为 1——↗
-
```

8.1.2 方向标志设定指令 CLD、STD

CLD、STD	设定方向标志
语法： CLD ←—— 将 D 标志清除为 0 STD ←—— 将 D 标志设定为 1 标志： D	

说明

D(Direction)方向标志会影响 CPU 处理字符串的方向。D 为1时是由高地址向低地址处理，D 为0时则是由低地址向高地址处理。为了避免在处理字符串时发生错误，我们通常会在处理之前先将方向标志设定为适当的值。

CLD(CLear Direction flag)会将 D 标志请除为0，DEBUG 把这个状态下的标志称为 UP，也就是处理字符串的方向是由低往高(地址)；而 STD(SeT Direction flag)则会将 D 标志设定为1，DEBUG 中称之为 DN(DownN)，意思是由高往低(地址)处理字符串。

实　例

```
-a
39E4:0100   STD     ；将 DF 设为1
39E4:0101   CLD     ；将 DF 清为 0
39E4:0102
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=39E4  ES=39E4  SS=39E4  CS=39E4  IP=0100   NV UP EI PL NZ NA PO NC
39E4:0100   FD            STD
-t
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=39E4  ES=39E4  SS=39E4  CS=39E4  IP=0101   NV DN EI PL NZ NA PO NC
39E4:0101   FC            CLD                      ↖—— DF 设定为 1
-t
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=39E4  ES=39E4  SS=39E4  CS=39E4  IP=0102   NV UP EI PL NZ NA PO NC
39E4:0102   B98AFF        MOV     CX,FF8A          ↖—— DF 清除为 0
```

8.1.3 中断标志设定指令 CLI、STI

CLI、STI	设定中断标志
语法：　CLI ←——— 将 I 标志清除为 0 　　　　STI ←——— 将 I 标志设定为 1 标志：　I	

说明

当 I 标志被设定为1时，CPU 可接受任何中断请求并加以处理；如果 I 标志被清除为0时，则软件中断请求会被屏蔽而变成无效。注意，硬件中断是无法屏蔽的（Non-Maskable），例如从键盘或鼠标输入数据时所产生的中断请求。

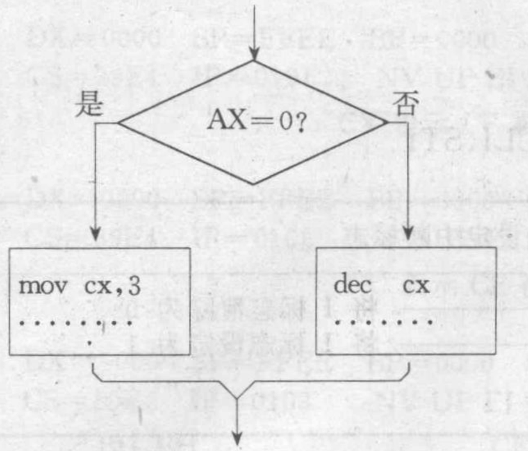
CLI（CLear Interrupt flag）会将 I 标志清除为0（DEBUG 中称为 DI，即 Disable Interrupt），而 STI 则会将 I 标志设定为1（DEBUG 中称为 EI，即 Enable Interrupt）。

实　例

```
-a
39E4:0100   CLI          ；将 IF 清为 0
39E4:0101   STI          ；将 IF 设为 1
39E4:0102
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=39E4  ES=39E4  SS=39E4  CS=39E4  IP=0100   NV UP EI PL NZ NA PO NC
39E4:0100  FA           CLI
-t
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=39E4  ES=39E4  SS=39E4  CS=39E4  IP=0101   NV UP DI PL NZ NA PO NC
39E4:0101  FB           STI                      ↖—— IF 清除为 0
-t
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=39E4  ES=39E4  SS=39E4  CS=39E4  IP=0102   NV UP EI PL NZ NA PO NC
39E4:0102  B98AFF       MOV    CX,FF8A           ↖—— IF 设定为 1
-
```

8.2 比较指令

当我们在控制程序流程时，常需要做一些判断。例如：如果 AX 为0则执行某段程序，如果不为0则执行另外一段程序。



CMP 指令便是用来比较两个操作数的大小(大于、小于、或相等),并根据结果设定标志值,以供后续的流程控制之用。

CMP	比较 OP1 与 OP2 的值
语法: CMP r/m, r/m/data ←—— OP1 与 OP2 不可同时为 m,且长度必须一致	
影响标志: C、P、A、Z、O	

说明

CMP(CoMPare)的意义是比较两个操作数的值,其动作与 SUB 指令一样,是由操作数1减掉操作数2,然后根据相减的结果来设定标志(C、P、A、Z、S、O),以便后续程序依标志值来决定程序执行的流程。注意,相减后的结果并不会存入 OP1,因此两个操作数的值都不会改变。

实 例

```
-a
21D0:0100   MOV  AL,20  ←———将 20h 存入 AL
21D0:0102   CMP  AL,30  ←———将 AL 值与 30h 做比较
21D0:0104   INT  3
21D0:0105
-t
    ↙——比较前 AL 为20
AX=0020  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=21D0  ES=21D0  SS=21D0  CS=21D0  IP=0102   NV UP EI PL NZ NA PO NC
21D0:0102   3C30        CMP     AL,30
-t
    ↙——比较后 AL 的值不变
AX=0020  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=21D0  ES=21D0  SS=21D0  CS=21D0  IP=0104   NV UP EI NG NZ NA PE CY
21D0:0104   CC          INT     3                        ↑  ↑        ↑
-
                         AL-30h 结果为负值(NG)
                         AL-30h 结果不为零,所以不相等(ZF=0,NZ)
                         AL 小于 30h,因此比较后产生借位(CY)
```

8.3　跳转指令

跳转指令可分为两种：

1)无条件跳转指令：不做任何条件的判断而直接跳转。

2)条件跳转指令：会先行判断某种条件是否成立，然后决定是否跳转。

以下我们分别介绍。

8.3.1　JMP 无条件跳转指令

JMP	跳往指定地址执行
语　法：	JMP　地址

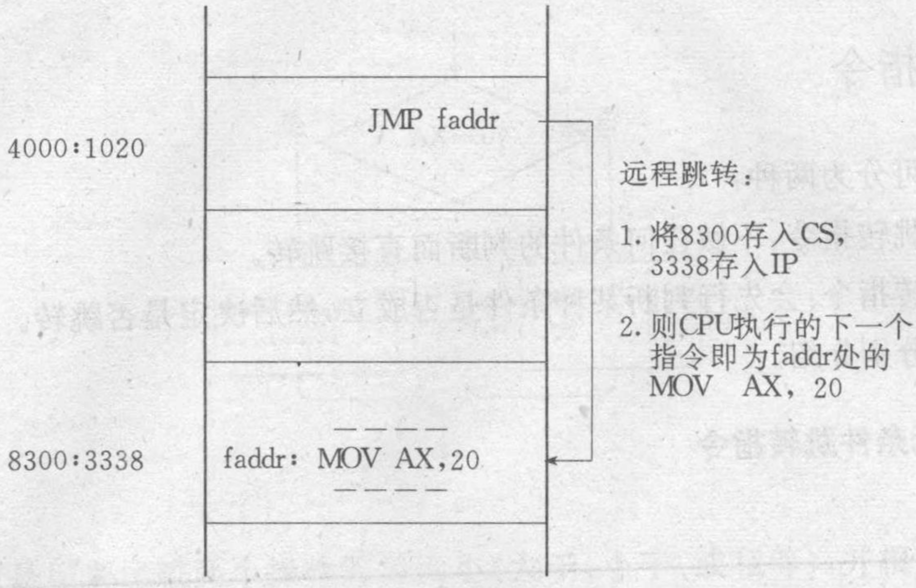
说明

JMP 之所以被称为无条件跳转，就是因为执行跳转后 CPU 会无条件转移到指定地址去执行另外一段程序，举例来说：

```
            JMP   TARGET ──────┐ 会跳转到 TARGET
            ............       │ 去执行
            ............       │
TARGET:     MOV   AX,0   ◄─────┘
            ............
            ............
```

JMP 指令跳转的范围如果是在一个段之内，则 JMP 会把目的地址的段内偏移量存入 IP，以产生跳转动作，这种段内的跳转方式称为“近程跳转”(Near Jump)。

如果 JMP 跳转范围跨越目前段边界，则 JMP 会把目的地址的“段地址：段内偏移量”存入 CS：IP 中，以产生跳转动作。像这种跨越多个段的跳转方式，我们称为“远程跳转”(Far Jump)。



实　例

```
-a
39C6:0100  JMP   105  ←————直接跳到地址 105 处执行
39C6:0102  CLC
39C6:0103  CLC
39C6:0104  CLC
39C6:0105  INT   3
39C6:0106
-t  ←——执行看看
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=39C6  ES=39C6  SS=39C6  CS=39C6  IP=0105   NV UP EI PL NZ NA PO NC
39C6:0105  CC           INT     3
-  ↖——果然跳到 105
```

8.3.2　Jxx 条件跳转指令

Jxx	当特定条件成立则跳往指定地址执行
语　法：　Jxx　地址	

说明

Jxx 称为条件跳转指令(Conditional Jump)，它是根据特定标志值或 CX 寄存器的值来判断是否执行跳转动作。共有 JA、JAE、JB、JBE、JC、JCXZ、JE、JG、JGE、JL、JLE、JNA、JNAE、JNB、JNBE、JNC、JNE、JNG、JNGE、JNL、JNLE、JNO、JNP、JNS、JNZ、JO、JP、JPE、JPO、JS、JZ 等三十几种，其中：

表8.1 条件跳转指令的英文符号

A	：Above，当 c=0、z=0时成立
B	：Below，当 c=1时成立
C	：Carry，当 c=1时成立
CXZ	：CX 寄存器的值为 0 (Zero)时成立
E	：Equal，当 z=1时成立
G	：Greater(大于)，当 z=0且 s=0时成立
L	：Less(小于)，当 s≠0时成立
N	：Not(相反条件)，需和其它符号配合使用
O	：Overflow，O=1时成立
P	：Parity，p=1时成立
PE	：Parity Even，p=1时成立
PO	：Parity Odd，p=0时成立
S	：Sign，s=1时成立
Z	：Zero，z=1时成立

我们通常会先做一些运算，例如 ADD、CMP 等，然后再以 Jxx 来判断标志的状况，并决定是否需要跳转到指定的地址去执行程序。如果是以 CMP 或 SUB 来做运算，则上表中的这几个符号很有用：

表8.2 与 CMP(SUB) 配合的跳转指令符号

E	：Equal，等于
A	：Above，无符号数的大于
B	：Below，无符号数的小于
G	：Greater，有符号数的大于
L	：Less，有符号数的小于
N	：Not，相反条件，需和其它符号配合使用

例如：

```
        CMP   CX,20
        JNE   Addr      ；NE 表示 Not Equal，即如果 CX 不等于 20
                        ；则跳至 Addr 去执行
        SUB   CX,10
Addr:   INC   CX,0
```

除了 N 外，其他的许多符号也可以相互搭配使用，例如：

```
        CMP   AX,15
        JAE   Addr2      ；AE 即大于或等于之意，如果 AX≥15
                         ；则跳至 Addr2 去执行
```

同理 JNAE 就是非大于或等于，也就是小于的意思。其它 Jxx 条件跳转指令依此类推。

除了比较大小的符号外，还有一些 Jxx 是按照某标志值或 CX 的值来判断，例如 JZ 即是当 ZF=1时跳转，JCXZ 则是当 CX 为0(Zero)时跳转。虽然说表7-1中的 E、A、B、G、L 等符号可用来判断有符号数或无符号数的大小，但事实上它们还是用标志值来做判断的。以 JE 指令来说，其中 E 代表 Equal(即 ZF=1时成立)，因此 JE 与 JZ 的作用是完全一样的。

又如 JAE 中的 AE 代表 Above(CF=0且 ZF=0)或 Equal(ZF=1)，由于 AE 的条件中有"ZF=0或 ZF=1"，因此 Z 标志为0或1均可，只要 CF=0时条件就成立。由此看来，JAE 和 JNC 的作用是一样的(CF=0时成立)。

实 例

```
-a
39C6:0100  MOV   AL,80
39C6:0102  CMP   AL,79
39C6:0104  JE   10C  ←——如果相等则跳转
39C6:0106  JG   10C  ←——以有符号数方式比较,如果大于则跳转
39C6:0108  JA   10C  ←——以无符号数方式比较,如果大于则跳转
39C6:010A  JMP   10C
39C6:010C  INT   3
39C6:010D
-g104
AX=0080  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=39C6  ES=39C6  SS=39C6  CS=39C6  IP=0104   OV UP EI PL NZ AC PO NC
39C6:0104  7406          JZ      010C  ←——不成立,80≠79
-t
AX=0080  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=39C6  ES=39C6  SS=39C6  CS=39C6  IP=0106   OV UP EI PL NZ AC PO NC
39C6:0106  7F04          JG      010C  ←——不成立,以有符号数比较时 80<79
-t
AX=0080  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=39C6  ES=39C6  SS=39C6  CS=39C6  IP=0108   OV UP EI PL NZ AC PO NC
39C6:0108  7702          JA      010C  ←——成立,以无符号数比较时 80>79
                                         所以跳到10C 执行
-t
AX=0080  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=39C6  ES=39C6  SS=39C6  CS=39C6  IP=010C   OV UP EI PL NZ AC PO NC
39C6:010C  CC            INT      3
-    ↖——果然由 108 直接跳到 10C(并未执行地址 10A 处的指令)
```

8.4 循环指令

循环指令可以让 CPU 多次重复执行某段程序,这样做可以简化程序的复杂程度。下面,我们就来介绍处理循环的三种指令:LOOP、LOOPE(Z)、LOOPNE(Z)。

8.4.1 LOOP 指令

LOOP、LOOPE(Z)、LOOPNE(Z)	循环指令集
语　法:　LOOP　　　地址 　　　　　LOOPE(Z)　地址 　　　　　LOOPNE(Z) 地址 影响标志:　无	

80x86的 LOOP 指令提供了高级语言的循环功能。LOOP 指令以 CX 为计数器,我们可以在 CX 中设定循环的次数,每执行循环一次,CX 的值会减1,直到 CX 值为0,循环停止。例如:

```
          MOV   AX,0
          MOV   CX,20
NEXT:     INC   AL  <-------------┐ 循环执行20次后(CX=0时)中止,
          ADD   AH,2              │
          LOOP  NEXT -------------┘ 然后再执行下一行程序
          CMP   AX,2814h
```

LOOPE(或 LOOPZ)除了以 CX 为计数器外,同时还测试 Z 标志的值,如果 CX=0或 Z=0,则循环停止。LOOPNE(或 LOOPNZ)则是在 CX=0或 Z=1时停止循环动作。例如:

```
          MOV   CX,20
NEXT:     DEC   AX  <-------------┐ 循环执行至 CX=0 或
          CMP   AX,5              │
          LOOPNE NEXT ------------┘ AX=5 时中止,
```

其实 LOOPNE 就是"不相等时继续循环"的意思,除了检查计数器 CX 的值之外,当 CMP 比较的结果相等时也会跳离循环。

另外,所有循环的范围均必须限制在128字节以内,也就是:

```
Lab:  MOV  AX,0    ┐
      ..........   │ 这一部分程序代码长度
      ..........   │
      ..........   │ 不能超过 128 个字节
      LOOP  Lab    ┘
```

实 例

```
-a
39C6:0100   MOV   AX,0
39C6:0103   MOV   CX,5   <——设定循环次数为 5
39C6:0106   INC   AX<-----┐
                          │ 形成循环,每次 AX 加 1
39C6:0107   LOOP   106----┘
39C6:0109   INT   3
39C6:010A
-g107   <——执行至 LOOP 处
AX=0001  BX=0000  CX=0005  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=39C6  ES=39C6  SS=39C6  CS=39C6  IP=0107   NV UP EI PL NZ NA PO NC
39C6:0107   E2FD          LOOP     0106
-t
                          ——每执行循环一次,CX 值就减 1

AX=0001  BX=0000  CX=0004  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=39C6  ES=39C6  SS=39C6  CS=39C6  IP=0106   NV UP EI PL NZ NA PO NC
39C6:0106   40            INC      AX
-g
      ——最后 AX=5          ——CX=0

AX=0005  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=39C6  ES=39C6  SS=39C6  CS=39C6  IP=0109   NV UP EI PL NZ NA PE NC
39C6:0109   CC            INT      3
-
```

8.5 有关子程序的指令

在编写程序时，我们通常会将程序中经常重复使用的部分写成子程序（Procedure 或称 Subroutine）。利用子程序，可将一个大的程序切割成许多小程序，每个小程序各负责一项功能并自成单元，例如：

主　程　序
字符串输入子程序
字符串输出子程序

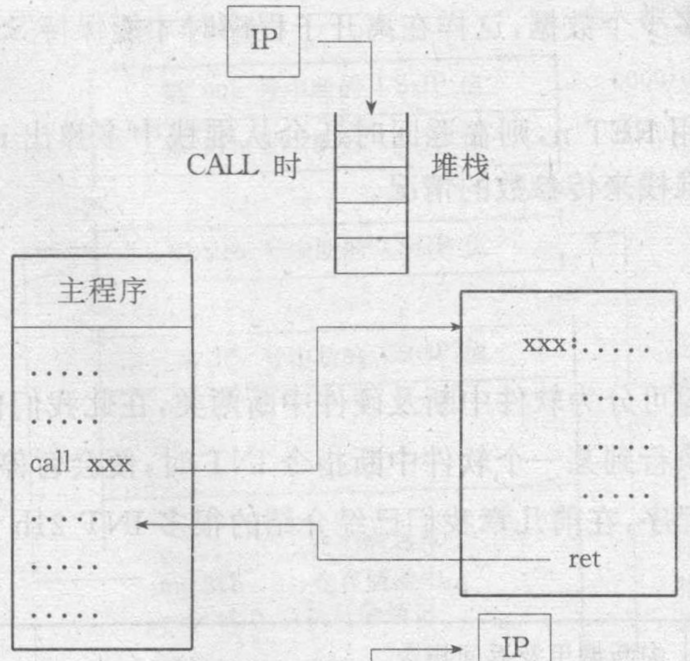
由于每一个子程序仅负责一个功能，因此大幅度降低了程序调试及维护的困难度。另外，每个子程序都可以相互调用并一再重复使用，所以我们将这些子程序称为“模块”（Module）；而将一个大程序划分成许多小模块的步骤，则称为“模块化”（Modulize）。

使用子程序的最大好处是，使程序代码可以重复运用而不必每次都重写，这就好比堆积木一样，我们仅需要利用几种基本的积木（子程序），便可重新组合出千变万化的物体（程序）。接下来，我们介绍调用子程序及返回主程序的指令：

CALL、RET	与子程序有关的指令
语　法：	CALL　地址 ←—— 调用指定地址上的子程序 RET ←—— 由子程序返回主程序 REN n ←—— 与RET相同，但从堆栈中多弹出n个字节
影响标志：	无

说明

CALL 指令会使程序转向指定的地址去执行。CALL 的动作与 JMP 有所不同，JMP 将程序转向指定的地址后，程序便无法返回；而 CALL 在转换执行地址之前，会把下一行指令的 IP 值压入堆栈（如果是跨段调用则压入 CS:IP 的值），而被调用的程序只要执行 RET 指令，便会将堆栈中的值弹回 IP（或 CS:IP），从而回到原程序继续执行：




```
-a
5B24:0100   MOV   AX,0003
5B24:0103   CALL   107
5B24:0106   INT   3
5B24:0107   ADD   AX,3
5B24:010A   RET
5B24:010B
-t
AX=0003  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=5B24  ES=5B24  SS=5B24  CS=5B24  IP=0103   NV UP EI PL NZ NA PO NC
5B24:0103   E80100         CALL      0107
-t
    返回地址(106)进栈,所以 SP 的值减2
AX=0003  BX=0000  CX=0000  DX=0000  SP=FFEC  BP=0000  SI=0000  DI=0000
DS=5B24  ES=5B24  SS=5B24  CS=5B24  IP=0107   NV UP EI PL NZ NA PO NC
5B24:0107   050300          ADD       AX,0003   ——程序转到地址107处去执行
-t
AX=0006  BX=0000  CX=0000  DX=0000  SP=FFEC  BP=0000  SI=0000  DI=0000
DS=5B24  ES=5B24  SS=5B24  CS=5B24  IP=010A   NV UP EI PL NZ NA PE NC
5B24:010A   C3              RET
-t
    返回地址被弹出堆栈,SP 的值加2
AX=0006  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=5B24  ES=5B24  SS=5B24  CS=5B24  IP=0106   NV UP EI PL NZ NA PE NC
5B24:0106   CC              INT       3         ——返回地址
-
```

在设计子程序时要特别注意:进入子程序时的 SS:SP 值,必须与离开时(执行 RET 时)完全一样,否则会造成子程序返回时弹出错误的返回地址。换句话说,在子程序的执行过程中,入

栈多少个数据就得弹出多少个数据，这样在离开子程序时才能保持 SS:SP 的值和进入子程序时一样。

如果在子程序中使用 RET n，则在返回时还会从堆栈中多弹出 n 字节的数据。这主要用于在调用子程序时通过堆栈来传参数的情况。

8.6 中断指令

中断按照产生的来源可分为软件中断及硬件中断两类，在此我们仅讨论软件中断。所谓的软件中断，就是当 CPU 执行到某一个软件中断指令 INT 时，便会暂停目前的工作而跳到指定的地址去执行中断服务程序，在前几章我们已经介绍的很多 INT 21h 中断就属于此类。

INT、IRET	中断调用及返回指令
语　法：	INT n ←— 产生中断，并指定要调用的中断服务程序编号 IRET ←— 由中断服务程序返回主程序
影响标志：	在执行 INT 指令时，CPU 会自动将标志寄存器的值入栈， 而在执行 IRET 时则会将堆栈中的标志值弹回标志寄存器

说明

INT n 会执行第 n 号中断服务程序，执行前 FLAG、CS、IP 会依次被压入堆栈，并且 I 标志会被清0，以防外部中断的干扰。

INT 之后必须跟随一个0～FFh 的数植，该数值就是中断服务程序的编号，譬如说我们经常使用的 INT 21h 就是让 CPU 暂停目前的工作转而去执行第21h 号的中断服务程序。

当执行一个中断命令时，CPU 会根据存储器中的一个对应表，我们称为中断向量表(Interrupt Vector Table)，找到该编号的中断服务程序所在地址，在未跳到该地址执行中断服务程序前，CPU 会自动将标志寄存器及中断指令下一行程序的段地址、段内偏移量依次压入堆栈，然后再转向中断服务程序。

每一个中断程序都以 IRET(Interrupt RETurn)指令做为结束，与 RET 类似，执行 IRET 后 CPU 会从堆栈区弹出三个字的数据，依序存入 IP、CS 和标志寄存器中，藉此返回原调用程序。调用中断服务程序的流程如图7-0所示。

中断向量表中每一个地址占四个字节(段地址:段内偏移量)，因此第0号中断服务程序的地址是在(00000～00003)，而第1号中断服务程序的地址是在(00004～00007)，依此类推，因此第 n 号中断服务程序地址是位于中断向量表的第4n 个字节上。

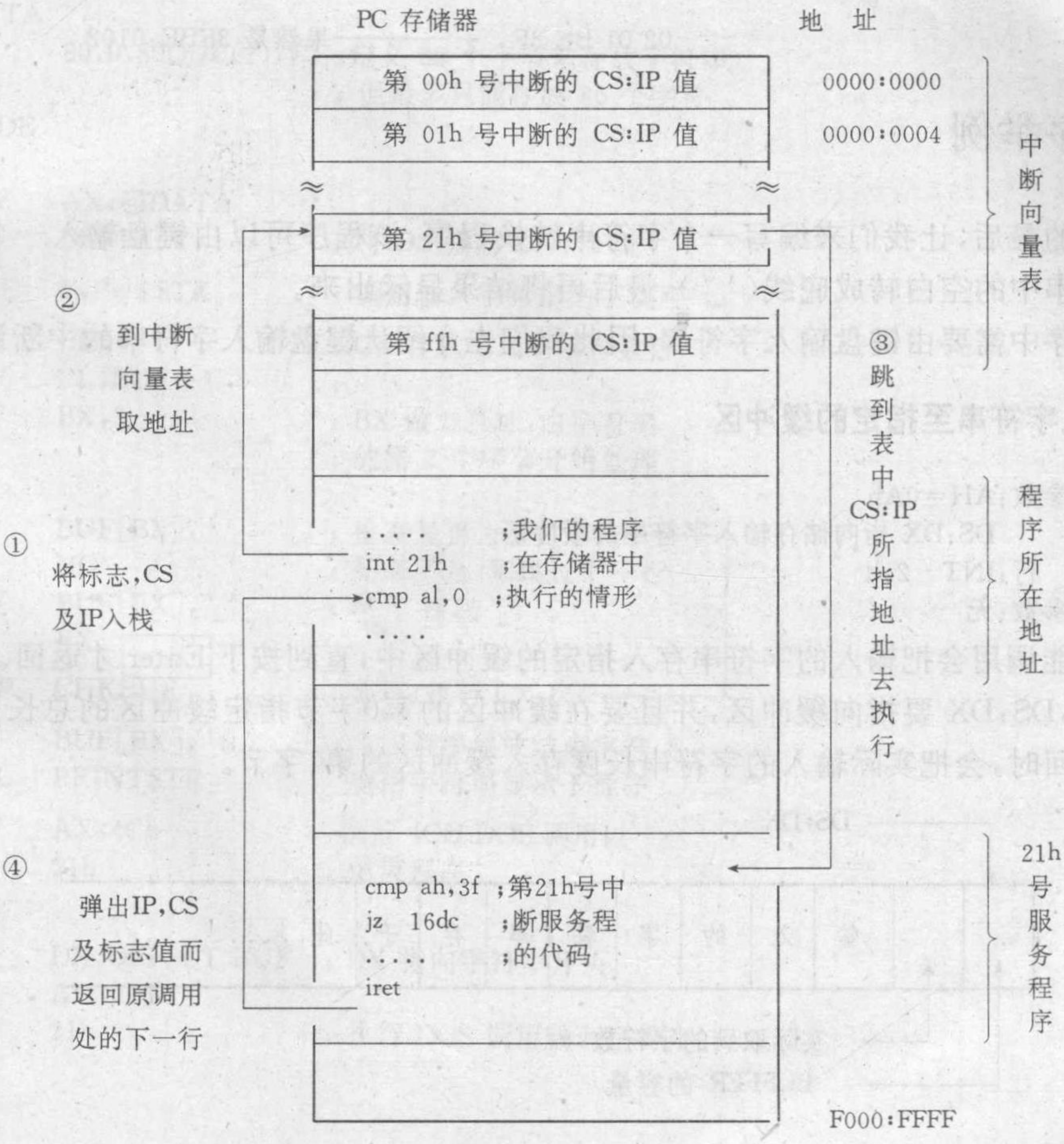


图8.1 中断处理的整个过程

实 例

```
-a
3FBE:0100   INT   21
3FBE:0102
-r
                    执行 INT 前的堆栈指针——
AX=0000   BX=0000   CX=0000   DX=0000   SP=FFEE   BP=0000   SI=0000   DI=0000
DS=3FBE   ES=3FBE   SS=3FBE   CS=3FBE   IP=0100     NV UP EI PL NZ NA PO NC
3FBE:0100   CD21            INT       21
-t ←————跟踪执行 INT 21h
          堆栈指针减少了 6(PUSH Flag、CS、IP)——
AX=0000   BX=0000   CX=0000   DX=0000   SP=FFE8   BP=0000   SI=0000   DI=0000
DS=3FBE   ES=3FBE   SS=3FBE   CS=D2FF   IP=16B4     NV UP DI PL NZ NA PO NC
D2FF:16B4   80FC3F          CMP       AH,3F
    ↖——↖跳转到 D2FF:16B4地址
-d 0:84 L4 ←————察看 INT 21h 的中断向量表(21h×4=84h)
0000:0080               B4 16 FF D2 ←———果然是 D2FF:16B4
-d ss:ffe8 L4 ←————察看堆栈顶端的4个字节是否为返回地址
```

```
3FBE:FFE0                 02 01 BE 3F  ←———果然是 3FBE:0102
```

8.7 程序举例

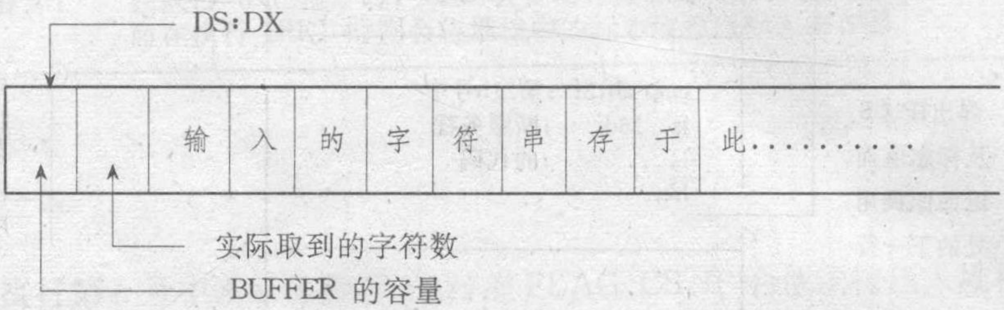
在本章的最后，让我们来编写一个字符串转换程序，该程序可以由键盘输入一个字符串，然后将字符串中的空白转成底线（'_'），最后再将结果显示出来。

由于程序中需要由键盘输入字符串，因此我们先介绍从键盘输入字符串的中断调用。

8.7.1 输入字符串至指定的缓冲区

输入参数：AH＝0Ah
　　　　　DS:DX 指向储存输入字符串的缓冲区
执　　行：INT　21h
返回参数：无

0Ah 功能调用会把输入的字符串存入指定的缓冲区中，直到按下 Enter 才返回。0Ah 功能调用执行时，DS:DX 要指向缓冲区，并且要在缓冲区的第0字节指定缓冲区的总长度。当0Ah 功能调用返回时，会把实际输入的字符串长度存入缓冲区的第1字节。



所以输入的字符串实际上是从缓冲区的第2字节存起。缓冲区的地址与容量由程序设计者自己指定，如输入的字符串超过缓冲区容量，则喇叭会发出声音。

使用方法

```
        .DATA
BUF     DB   80,0,80 DUP(?)     ；定义 82 个字节来存放字符串
                                ；但最多只能存放 80 个字符
        .CODE
        MOV   AX,@DATA
        MOV   DS,AX
        MOV   DX,OFFSET BUF     ；令 DS:DX 指向 BUF
        MOV   AH,0Ah
        INT   21H
        ……
        ……
```

8.7.2 程序举例

PRG8-1. ASM

```
.286                            ；设定汇编使用286指令
.MODEL SMALL                    ；设定为 Small 存储器模式
```

```
        .DATA
BUF     DB      80,0,80DUP(?)       ;定义 82 个字节来存放字符串
                                    ;但最多只能存放 80 个字符
        .CODE
BEGIN:
        MOV     AX,@DATA
        MOV     DS,AX               ;将 DS 指向 DATA 段
        CALL    INPUTSTR            ;调用输入字符串的子程序
        MOV     CH,0                ;CX 存放输入的字符数目
        MOV     CL,BUF+1            ;
        MOV     BX,2                ;BX 做为基址,由字符串
                                    ;的第 2 个字节开始处理
CHKBUF:
        CMP     BUF[BX],' '         ;检查是否为空白字符←
        JNE     NO                  ;如果不是,则跳过下一行
        MOV     BUF[BX],'_'         ;将' '换成'_'
    NO: INC     BX
        LOOP    CHKBUF              ;循环(执行 CX 次)
        MOV     BUF[BX],'$'         ;在字符串尾放结束字符 $
        CALL    PRINTSTR            ;调用字符串显示子程序
        MOV     AX,4Ch              ;执行 4CH DOS 调用以
        INT     21h                 ;结束程序
INPUTSTR:
        MOV     DX,OFFSET BUF       ;DX 指向字符串开头
        MOV     AH,0Ah
        INT     21h                 ;执行 DOS 调用输入字符串
        RET
PRINTSTR:
        MOV     BUF,13              ;将 BUF 最前面的两个字节
        MOV     BUF+1,10            ;换成回车(CR)及换行(LF)
        MOV     DX,OFFSET BUF       ;DX 指向字符串开头
        MOV     AH,09h
        INT     21h                 ;执行 DOS 调用显示字符串
        RET
        .STACK
        END     BEGIN               ;程序结束
```

执行结果

```
C:\>prg8-1
MASM is easy to learn, if and only if you read this book. [Enter]
MASM_is_easy_to_learn,_if_and_only_if_you_read_this_book.
```

说明

本程序执行时的步骤如下:

1)首先调用中断子程序从键盘读取字符串。

2)将读入字符串的长度存入 CX,做为循环(LOOP)计数之用。

3)将 BX 设为2,用于索引字符串中的字符。

4)进入循环(执行 CX 次)。

A. 比较 BX 所指向的字符(BUF[BX])是否为'—'

B. 如果是,则换为'_'

C. 将 BX 加1,指向 BUF 中的下一字符

5)将输入字符串的尾端字符设为'$',然后调用 PRINTSTR 子程序显示字符串。

6)结束程序。

本章作业

1. 请解释下列名词

远程跳转 (Far Jump)

近程跳转 (Near Jump)

模块(Module)与模块化(Modulize)

2. 试说明下列指令的功能及语法格式?

STC、CLC、CMC、STD、CLD、STI、CLI、JMP、Jxx、CMP、LOOP、LOOPE、LOOPNE、CALL、RET、INT、IRET

3. 试写出15种条件跳转指令(Jxx),并说明其判断条件的方式。

4. 请写出下列程序执行中每一步骤的 CF、DF、IF 的值

	CF	DF	IF
STC	?	0	1
STI	?	?	?
CLC	?	?	?
CLD	?	?	?
CMC	?	?	?
CLI	?	?	?

5. 试说明 LOOP 指令与寄存器 CX 有何关系?

6. 请将下列程序执行的顺序以阿拉伯数字标在每行指令之前,并指出该段程序执行结束后 AL 的值。

```
        MOV   AL,8Fh
        CMP   AL,20h
        JG    EXIT1
        CMP   AL,70h
        JA    EXIT2
        MOV   AL,01
        JMP   END
EXIT1:
        MOV   AL,02
        JMP   END
EXIT2:
        MOV   AL,03
END:    RET
```

7. 请比较下列两个程序执行结束后的 CX 及 AX 值，并说明原因？

```
      MOV  CX,20                     MOV  CX,20
      MOV  AX,20                     MOV  AX,20
SUB1:                           SUB1:
      SUB  AX,2                      SUB  AX,2
      CMP  AX,0                      CMP  AX,0
      LOOP SUB1                      LOOPNE SUB1
```

8. 请写出主程序调用子程序及 INT 21h 时的 SP 值？

```
主程序
-----------------------------------------------
.....
.CODE
..........
MOV  AL,99                          SP=FFEEH
CALL  FUNCT1
MOV  AL,0Bh
INT  21h
...........
.......
FUNCT1:
         AAM                        SP=?
         OR  AX,3030h
         RET
..........
```

```
INT 21h 中断服务程序
-----------------------------------------------
CMP  AH,3Fh                         SP=?
...........
.........
IRET
```

9. 请改写 PRG8-1. ASM，使程序可以将输入字符串中的小写字母全部转成大写。

10. 试编写一个程序可以将输入字符串中的字母以相反顺序显示出来，例如输入"ABC"则显示出"CBA"。

11. 试编写一个程序可以做2位正整数的加法，例如输入"12＋6"，则输出"18"。

12. 请将上一题改为可以处理2位正整数的加法与减法。

第九章　程序流程控制伪指令

上一章中我们虽然已介绍了一些流程控制指令，但是对于一些较复杂的程序流程控制，使用这些指令可能会让程序变得很长且不容易阅读。因此，MASM 另外提供了一系列的伪指令来简化程序流程控制，它们包括：

□IF…ELSEIF…ELSE…ENDIF 伪指令

□WHILE…ENDW、REPEAT…UNTIL、BREAK、CONTINUE 伪指令

当我们要处理一些较复杂的流程控制时，使用一大群的比较指令或条件跳转指令会十分吃力，此时可通过流程控制伪指令来加以简化，并增加程序的可读性。

9.1　IF…ELSEIF…ELSE…ENDIF 伪指令

9.1.1　IF…ENDIF 流程结构

.IF 用做条件判断，它必须配合.ENDIF 一起使用，其使用格式如下：

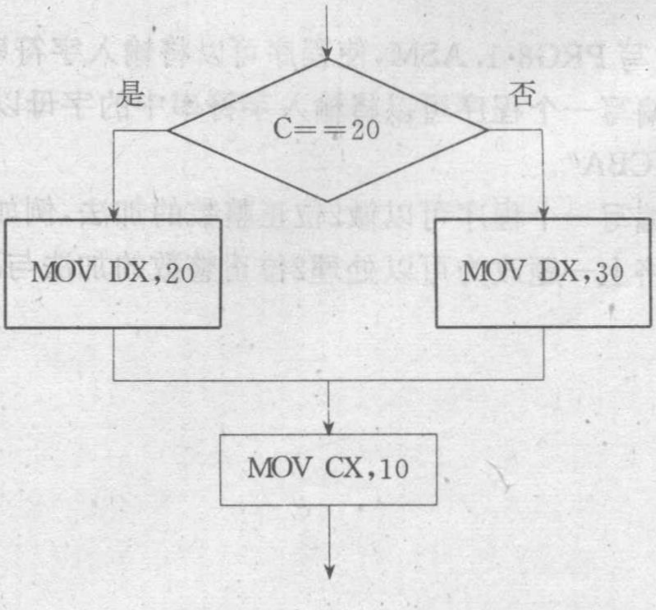
```
.IF 条件表达式 A
        MOV   AX,20     } 条件 A 成立则执行该部分程序。
        MOV   BX,30     } 否则跳到 .ENDIF 的下一条程序
.ENDIF
MOV   CX,40
........
```

上例中当条件表达式 A 成立时，程序会接著执行 MOV AX,20和 MOV BX,30这两行程序，否则程序会自动跳到.ENDIF 的下一行程序(MOV CX,40)。

9.1.2　IF…ELSE…ENDIF 流程结构

当遇到程序需要分别处理条件表达式成立与否的情况时，我们可以使用 IF…ELSE…ENDIF 的程序流程结构，例如：

```
......
.....
.IF   CX==20
      MOV DX,20     ←── 第1部分
.ELSE
      MOV DX,30     ←── 第2部分
.ENDIF
MOV CX,10
.....
```



注：'==' 代表相等，在伪指令中要用'=='，不可以用'='

在这个例子中，如果 CX 的值是20，则 DX 会存入20；如果 CX 的值不等于20，那么 DX 就会存入30。

实际上 MASM 是将上述程序做了以下的代换：

```
        .IF   CX==20
*               CMP   CX,014h       <——  CX 是否等于 14h（十进制的 20）
*               JNE    @C0001       <——  CX 如果不等于 20 则跳到@C0001处
                MOV   DX,20         <——  CX 如果等于 20 则将 20 存入 DX
        .ELSE
*               JMP   @C0003
*  @C0001:
                MOV   DX,30         <——  CX 不等于 20 时，将 30 存入 DX
        .ENDIF
*  @C0003:
```

注：汇编完成后，方框中的伪指令将被标有 * 的指令取代

在最前头标有 * 符号的几行程序，都是经过 MASM 替换的结果。相比较之下，使用流程控制伪指令的程序的确是比较简洁易懂！

9.1.3 条件表达式

在介绍.IF 时我们使用了很简单的条件表达式：(CX==20)，但是并未详细说明条件表达式的使用，以下是条件表达式的使用方法。

□将常数、变量、寄存器加上一些运算符号(Operator)就可以组成条件表达式，例如：(AX==3)、(MEM>=3)等等，伪指令条件表达式的运算符号如下。

表9.1 条件表达式运算符号

运算符号	含义
==	等于
!=	不等于
>	大于
>=	大于等于
<=	小于等于
&	与 TEST 指令相同，以位为单位做 AND 运算
!	逻辑 NOT 运算（参考第六章逻辑运算指令）
&&	逻辑 AND 运算（参考第六章逻辑运算指令）
||	逻辑 OR 运算（参考第六章逻辑运算指令）

□条件表达式本身就是一个运算式，如果运算式的值为0则表示假，否则为真(1、2、3…等非0的值均被视为真)。例如：

表9.2　条件表达式范例

当 AX=	判　　断 条件表达式	条件表达式是否成立 (1=成立，0=不成立)
6	.IF　AX	1
0	.IF　!AX	1
0	.IF　AX!=0	0
0	.IF　AX==0	1
5	.IF　AX==5	1
5	.IF　AX>4	1
6	.IF　AX&&0	0
6	.IF　AX‖0	1

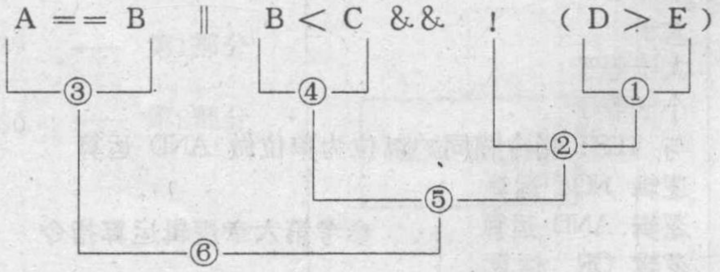
□各种运算符号的优先次序并不相同，其优先次序如下：

表9.3　运算优先次序

运算符号	优先次序
!	1
<,<=,>,>=,==,!=	2
&	3
&&	4
‖	5

运算符号的结合顺序是自左而右，而且可以另外加上括号。在括号内的条件表达式具有最高优先权，然后才依照上表的优先次序依序计算。例如：A==B‖B<C&&!(D>E)，其计算次序如下：

次序	条件表达式
1	D>E
2	!(D>E)
3	A==B
4	B<C
5	B<C&&!(D>E)
6	A==B‖B<C&&!(D>E)



注意

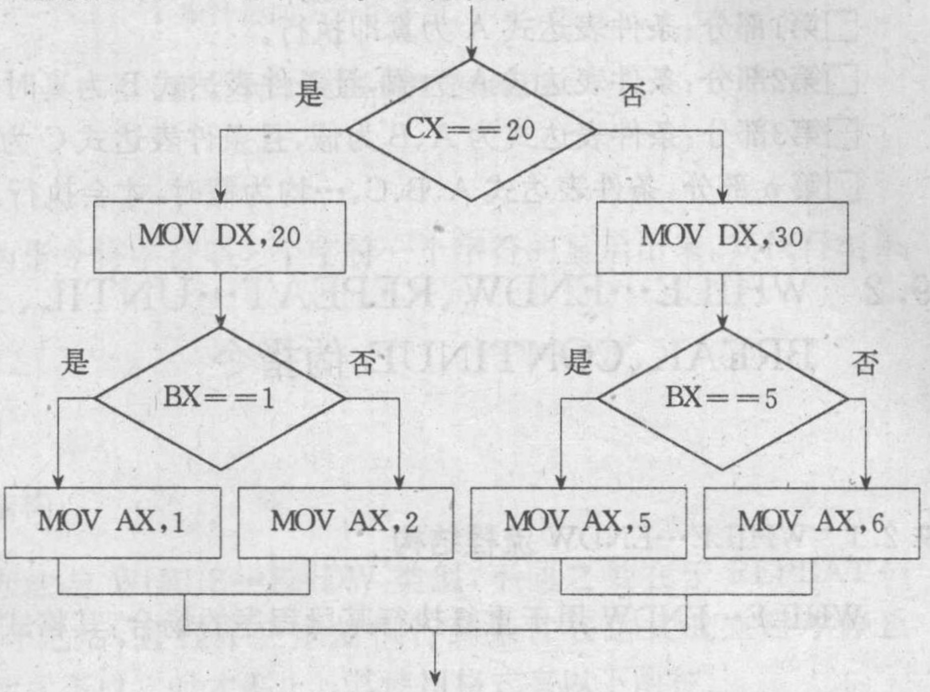
由于 MASM 对于条件表达式的判断是使用比较指令来替换的，而且比较指令 CMP 和 TEST 的两个操作数是不能同时为立即数的，所以 MASM 无法对有些条件表达式进行替换，

所以也就不能使用。如条件表达式:1==1 和 0&1都是错误的写法。

9.1.4 .IF 嵌套

.IF 伪指令也可以多层嵌套,也就是说.IF 之下还可以有另外一层.IF:

```
.IF    CX==20
       MOV  DX,20
       .IF   BX==1
             MOV  AX,1
       .ELSE
             MOV  AX,2
       .ENDIF
.ELSE
       MOV  DX,30
       .IF   BX==5
             MOV  AX,5
       .ELSE
             MOV  AX,6
       .ENDIF
.ENDIF
```

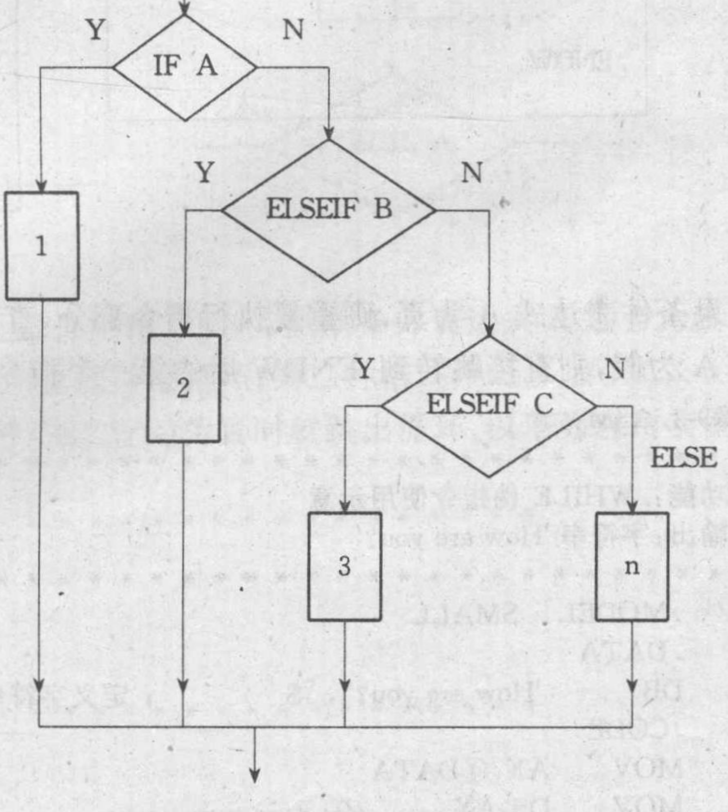


在 MOV DX,20和 MOV DX,30之后的是第二层的.IF,而且第二层的下还可以有第三层的.IF……。

9.1.5 IF…ELSEIF…ELSE…ENDIF 结构

当需要同时判断两个以上的条件表达式时,我们可以使用.IF 的多重选择格式:

```
.IF 条件式 A
      指令 1
      指令 2    } 第 1 部分
      ........
.ELSEIF 条件式 B
      指令 1
      指令 2    } 第 2 部分
      ........
.ELSEIF 条件式 C
      指令 1
      指令 2    } 第 3 部分
      ........
-------------------------
可以有多个 .ELSEIF
-------------------------
.ELSE
      指令 1
      指令 2    } 第 n 部分
      ........
.ENDIF
```



程序会依次检查条件 A、B……直至条件成立，并执行该条件下的程序，然后再跳转到. ENDIF 的下一行程序。. ELSEIF 可随着判断条件的多少而增减，一个. IF 可以配有多个. ELSEIF 或没有. ELSEIF。由于. ELSE 是在前面所有条件都不成立时才会执行，因此它必须写在所有. ELSEIF 之后(. ELSE 如果不需要可以省略)。程序在执行时的详细流程如下：

□第1部分：条件表达式 A 为真即执行。

□第2部分：条件表达式 A 为假，且条件表达式 B 为真时才会执行。

□第3部分：条件表达式为 A、B 为假，且条件表达式 C 为真时才会执行。

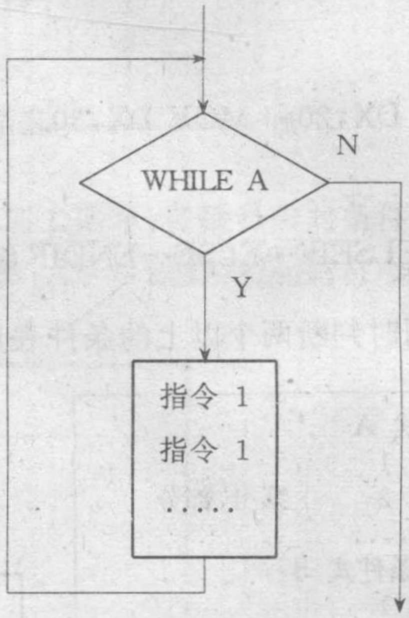
□第 n 部分：条件表达式 A、B、C、…均为假时，才会执行。

9.2 WHILE…ENDW、REPEAT…UNTIL、BREAK、CONTINUE 伪指令

9.2.1 WHILE…ENDW 流程结构

WHILE…ENDW 用于重复执行某段程序的场合，其格式如下：

```
.WHILE 条件表达式 A
    指令 1  ┐
    指令 2  ├ 指令部分
    ……      ┘
.ENDW
```



如果条件表达式 A 为真，则重复执行指令部分，直到条件表达式 A 为假才停止；如果条件表达式 A 为假，则直接跳转到. ENDW 后的第一个指令执行，请看以下例子：

```
    PRG9-1.ASM
;*************************************
;  程序功能：.WHILE 伪指令使用示意
;  程序输出：字符串 'How are you?'
;*************************************
        .MODEL . SMALL
        .DATA
BUF1    DB      'How are you?','$'      ；定义字符串
        .CODE
BEGIN:  MOV     AX,@DATA
        MOV     DS,AX
```

```
MOV    BX,0                          ;将 0 存入 BX
.WHILE (BUF1[BX]!='$')               ;到'$'时结束
       MOV   DL,BUF1[BX]             ;取一个字符至 DL
       MOV   AH,02                   ;调用 02h 号 DOS 调用
       INT   21h                     ;显示该字符
       INC   BX                      ;指针指向下一个字符
.ENDW
MOV    AX,4C00h                      ;使用 4CH DOS 调用结束程序
INT    21h
.STACK
END BEGIN
```

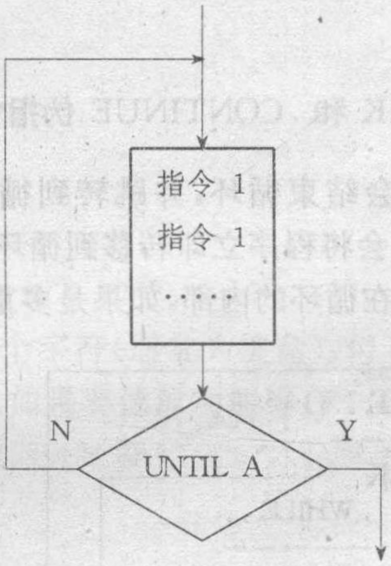
在上例中我们使用.WHILE 伪指令将字符串一个字符一个字符的显示出来。其执行结果如下：

```
C:\>PRG9-1
How are you?
C:\>_
```

9.2.2 REPEAT…UNTIL 流程结构

REPEAT…UNTIL 的使用方法与 WHILE…ENDW 类似，不同之处在于 REPEAT…UNTIL 的条件判断是在程序循环之后，且程序会重复执行到条件表达式成立后才停止(WHILE…ENDW 则是在条件表达式不成立时才停止)，其使用格式有以下两种。

```
.REPEAT
    指令 1  }
    指令 2  } 第一部分
    ……      }
.UNTIL 条件表达式 A
或是
.REPEAT
    指令 1  }
    指令 2  } 第二部分
    ……      }
.UNTILCXZ 条件表达式 B
```



在第一种格式中，程序在执行了第一部分的指令后，才判断条件表达式 A 是否为真，如果为假，则重复执行第一部分的指令，如果为真就结束循环；第二种格式则是多加了 CX==0这个条件表达式；也就是(CX==0)或(条件表达式 B)为真时就跳出循环。以下为应用实例。

```
  PRG9-2.ASM
;****************************************
;  程序功能:.REPEAT 伪指令使用示意
;  程序输入:从键盘输入字符串,直到输入
;          '$'或字符串长度超过80为止
;  程序输出:显示字符串
;****************************************
        .MODEL  SMALL
        .DATA
BUFFER  DB  80 DUP(0),'$'              ;该缓冲区用于储存输入的字符串
```

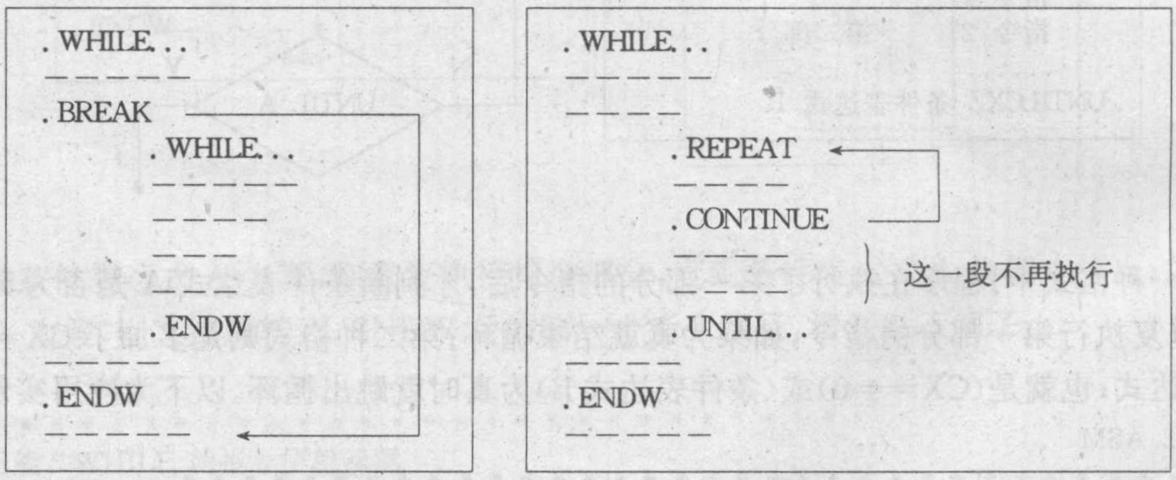
```
         .CODE
BEGIN:   MOV    AX,@DATA
         MOV    DS,AX
         MOV    BX,0                          ;将 0 存放 BX
         .REPEAT
                MOV  AH,01                    ;执行 01h 号 DOS 功能调用
                INT  21h                      ;从键盘输入字符
                MOV  BUFFER[BX],AL            ;将字符送入缓冲区
                INC  BX                       ;指针指向缓冲区下一个位置
         .UNTIL (AL=='$'||BX==80)  ;直到 $ 被键入为止
                                              ;或输入字符串的长度超过80为止
         MOV    DX,OFFSET BUFFER              ;DX 缓冲区
         MOV    AH,09h                        ;执行 09h 号 DOS 功能调用
         INT    21h                           ;以显示字符串
         MOV    AX,4C00h                      ;使用 4CH DOS 调用结束程序
         INT    21h
         .STACK
         END BEGIN
```

在这个程序中，我们使用.REPEAT 和.UNTIL 所构成的循环来读取输入字符，直到输入的字符为"$"或字符串长度为80后，再将 Buffer 中的字符显示出来。其执行结果如下：

```
C:\>PRG9-2
Hello world $ Hello world
用户输入此部分
C:\>_
```

9.2.3 .BREAK 和 .CONTINUE 伪指令

.BREAK 会结束循环，并跳转到循环后(.ENDW 或.UNTL)的指令去开始执行，而.CONTINUE 则会将程序立即转移到循环的开头(.WHILE 或 REPEAT)。.BREAK 和.CONTINUE 只能用在循环的内部，如果是多重循环时，这两个指令只会影响指令所在的循环，如下图。



.BREAK 和.CONTINUE 伪指令也可以和.IF 合起来使用，其格式如下：

```
.BREAK  .IF 条件表达式
.CONTINUE  .IF 条件表达式
```

下面这个例子，只能输入 '0' 到 '9' 之间的字符，并且必须在按下 Enter 键后结束，所以使用.BREAK .IF 和.CONTINUE .IF 是最方便的：

```
.WHILE  1                          ；无限循环
      MOV AH,08h                   ；使用 DOS 调用输入字符
      INT  21h                     ；但不在屏幕上显示(下节介绍)
      .BREAK .IF AL==13            ；如果键入的是[Enter]
                                   ；则跳出循环
      .CONTINUE .IF (AL<'0')||(AL>'9')  ；如果不是数字
                                   ；0～9 则返回循环的开头
      MOV DL,AL                    ；
      MOV AH,02h                   ；
      INT  21h                     ；显示字符
.ENDW
```

上例中我们使用了 WHILE 1来作为重复循环的条件判断，由于常数1永远为真，因此这个循环会一直重复执行，直到.BREAK .IF 后的条件表达式成立为止。

请注意!在使用.BREAK .IF 和.CONTINUE .IF 时，并不需要以.ENDIF 来作为结束。

9.3 程序举例

本节实例会读取方向键，并根据所按的方向键来移动光标，直到按下[Enter]键才结束。为了便于说明，以下先介绍这个程序所用到的中断服务程序。

9.3.1 等待并读取按键数据

输入参数：AH=08h
执　　行：INT　21h
返回参数：AL=所读取的字符

说明

08h 调用会等待用户由标准输入装置输入一个字符(通常为键盘)，但不在屏幕上显示出来，然后将这个字符的 ASCII 码存入 AL 寄存器。如果要读取功能键(F1、F2…或方向键等)，则必须调用这个中断程序两次，第一次 AL 会先返回00h，第二次才会返回功能键扫描码。

9.3.2 读取屏幕光标位置

输入参数：AH=03h
　　　　　BH=当前显示页号
执　　行：INT　10h
返回参数：DH=光标的纵坐标(第几行)
　　　　　DL=光标的横坐标(第几列)

说明

INT 10h 是处理屏幕显示的中断，它包括一些绘图及显示字符的功能调用，其中03h 是用来读取当前光标位置的服务程序。由于在文字模式下，BIOS 可同时保存好几个显示页面的数据，因此我们还必须将当前显示页号存在 BH 中(一般状况下均是使用第0页，因此我们将 BH 设为0)。执行调用后，光标的纵坐标及横坐标会分别由 DH、DL 返回。

9.3.3 设定屏幕光标位置

输入参数:AH=02h

BH=当前显示页号

DH=光标的纵坐标(第几行)

DL=光标的横坐标(第几列)

执　　行:INT　10h

返回参数:无

说明

与上述的03h 功能调用相反,02h 调用是用来设定光标的显示位置,因此我们必须先将显示的页号、光标的纵、横坐标分别存放在 BH、DH、DL 这三个寄存器中,然后再调用此服务程序。

9.3.4 程序 PRG9-3. ASM

```
        .8086
        .MODEL   SMALL
        .DATA
CUR_X   DB       0                              ; 储存光标的横坐标
CUR_Y   DB       0                              ; 储存光标的纵坐标
        .CODE
BEGIN:  MOV      AX,@DATA                       ; 将 DS 指向数据段
        MOV      DX,AX
        MOV      AH,03                          ; 调用 INT 10h 第 03h 号功能
        MOV      BH,0                           ; 以取得当前光标位置
        INT      10H                            ; 执行中断 10h
        MOV      CUR_X,DL                       ; 保存横坐标
        MOV      CUR_Y,DH                       ; 保存纵坐标
        .REPEAT                                 ; 读取键盘输入数据循环开始
           MOV   AH,08                          ; 等待并读取按键数据
           INT   21h
        .IF (AL==00h)                           ; 如果是功能键
              MOV   AH,08                       ; 再读功能键的扫描码
              INT   21h
              .IF (AL==72)&&(CUR_Y!=0)          ; 当按[↑]且光标不在第一行
                DEC   CUR_Y                     ; 将纵坐标减 1
              .ELSEIF (AL==75)&&(CUR_X!=0)      ; 当按[←]且光标不在最左一列
                DEC   CUR_X                     ; 将横坐标减 1
              .ELSEIF (AL==77)&&(CUR_X!<79)     ; 当按[→]且光标不在最右一列
                INC   CUR_X                     ; 将横坐标加 1
              .ELSEIF (AL==80)&&(CUR_Y<24)      ; 当按[↓]且光标不在最后一行
                INC   CUR_Y                     ; 将纵坐标加 1
              .ELSE                             ; 如果是其它功能键
                .CONTINUE                       ; 则再读下一个键(跳转至 REPEAT)
              .ENDIF
              MOV   AH,02                       ; 调用 02h 功能以设定光标位置
              MOV   BH,0                        ; 当前显示页号设为 0
              MOV   DL,CUR_X                    ; 横坐标送入 DL
              MOV   DH,CUR_Y                    ; 纵坐标送入 DH
```

```
        INT    10h                              ;执行 10h 中断
      .ENDIF
    .UNTIL (AL==0Dh)                            ;当按键不是[Enter]则继续循环
    MOV   AX,4C00h
    INT   21h                                   ;结束程序
    .STACK
    END   BEGIN
```

程序说明

1)程序首先执行 INT 10h 的第03h 调用,以取得目前的光标位置并将它存入存储器。

2)处理按键的循环中,我们首先判断取得的 ASCII 码是不是00h,如果等于00h,则表明按下的是功能键,因此我们还必须再执行一次08h 调用,以取得功能键的扫描码,然后判断扫描码是否为上下左右方向键,并根据按键来改变光标位置。

3)如果取得的是一般按键,则再回到读取键盘循环。

4)如此周而复始直到按下[Enter]键为止。

本章习题

1.请判断下列程序执行结束后的 AX、CX 值?

```
MOV   AX,20
MOV   CX,30
SUB   AX,CX
.IF (AX==20) || (CX==30)
    .IF AX!=20
        SUB  AX,10
        ADD  CX,20
    .ENDIF
.ELSE
    SUB  AX,20
    SUB  CX,10
.ENDIF
```

2.请判断下列程序执行结束的 AX、CX 值?

```
MOV   CX,20
MOV   AX,20
MOV   BX,10
.REPEAT
    ADD  AX,BX
    INC  CX
    DEC  BX
.UNTIL  BX==0
```

3.请判断下列程序执行结束后的 AX、CX 值?

```
MOV   AX,20
MOV   CX,10
MOV   BX,2
MOV   DX,20
.WHILE 1
    ADD  AX,CX
    SUB  DX,BX
    .BREAK  IF  DX==0
```

```
    .CONTINUE  IF  DX!=10
    .ADD  BX,3
  .ENDW
```

4. 请说明在条件表达式中，运算符号的运算优先次序。

5. 请将下列运算式依运算先后次序列出。

   CX==10 && BX < 20 ‖ (DX > && AX != 40)

6. 请将下列条件叙述转换为由流程控制伪指令所构成的程序：

   当 CX 不等于 BX 且 DX 等于 1 或 AX 等于 30 时，则 SI 存入 30，否则将 0 存入 DI。

7. 请将下列条件叙述转换为由流程控制伪指令所构成的程序：

   重复将 1 加入 AX，然后再将 CX 减 1，直到 CX 等于 0 或 AX 等于 BX。

8. 请将下列条件叙述转换为由流程控制伪指令所构成的程序：

   当 AX 不等于 BX 且 DX 小于或等于 CX 时，则重复将 BX 加2，DX 减2直到不符合前述条件为止。

9. 改写 PRG8-1. ASM 将程序流程控制部分改成流程控制伪指令。

10. 改写 PRG9-3. ASM，除了可用方向键移动光标之外，也可以在光标处输入一般的字符。

11. 请编写一个程序，可不断由键盘读取输入，如果为一般字符则显示其 ASCII 码，如果为功能键则显示其扫描码，直到按下 Esc 键为止。

第十章　字符串操作指令

本章所要介绍的是字符串操作指令和重复前缀指令。内容包括：

□字符串传送指令(MOVSB、MOVSW、MOVSD)

□字符串比较指令(CMPSB、CMPSW、CMPSD)、字符串搜索指令(SCASB、SCASW)

□字符串载入指令(LODSB、LODSW)、字符串储存指令(STOSB、STOSW)

□前缀指令(REP、REPE、REPNE)

10.1　字符串操作指令总览

我们可以将字符串(Strimg)解释成一串连续的数据，其长度可以是字节、字、或是双字的倍数。80x86所提供的字符串操作指令集包含了 MOVS、CMPS、SCAS、LODS 及 STOS 等五大类，其中指令尾部的"S"表示字符串(String)的意思。当你想以字节为单位来处理字符串时，则可以在指令后面加上一个字母"B"(Byte)；若是以字来处理字符串时，就要在指令后面加上字母"W"(Word)；而以双字为单位来处理字符串时，则要在指令后加上字母"D"(DWord)。

因此，希望读者不要被一大堆看来类似的指令所迷惑了，下表为这些指令的分类：

表10.1　字符串处理指令的分类

指令种类	每一次处理字符串的长度		
	字节	字	双字
字符串传送指令	MOVSB	MOVSW	MOVSD
字符串比较指令	CMPSB	CMPSW	CMPSD
字符串搜索指令	SCASB	SCASW	SCASD
字符串载入指令	LODSB	LODSW	LODSD
字符串贮存指令	STOSB	STOSW	STOSD

双字类的字符串操作指令仅适用于386以上的 CPU，其中 SCASD、LODSD、和 STOSD 必须使用到 EAX 这个32位的寄存器，由于已超出本书的范围，所以这里不做介绍。

10.2　字符串传送指令

MOVSB、MOVSW、MOVSD	字串传送指令
语　法：MOVSB ←—— 将DS:SI地址的一个字节数据传送到ES:DI，并将SI及DI加1(或减1，视方向标志值而定) MOVSW ←—— 将DS:SI地址的一个字的数据传送到ES:DI，并将SI及DI加2(或减2，视方向标志值而定) MOVSD ←—— 将DS:SI地址的一个双字的数据传送到ES:DI，并将SI及DI加4(或减4，视方向标志值而定)	
影响标志：无	
注：方向标志DF=1时，每传送一个字节(或字)后，SI和DI会被减去1(或2)	

说明

字符串传送指令是将字符串从存贮器的某一个位址传送到另一个位址上。MOVSB (MOVe String Byte by byte)、MOVSW (MOVe String Word by word)、和 MOVSD (MOVe String Double word by double word)都是将存在 DS:SI 地址上的源字符串传送至 ES:DI 地址。

方向标志 DF 会影响到字符串传送的次序,当 DF=0时,是由低地址往高地址的方向逐一传送字符串,因此每执行一次 MOVSB 后,SI 和 DI 会分别加1(MOVSW 则会使 SI 和 DI 分别加2,MOVSD 则会使 SI 及 DI 分别加4),以指向字符串的下一个字符。而在 DF=1时,每执行一次 MOVSB,则 SI 和 DI 会被减去1(或2)。例如:

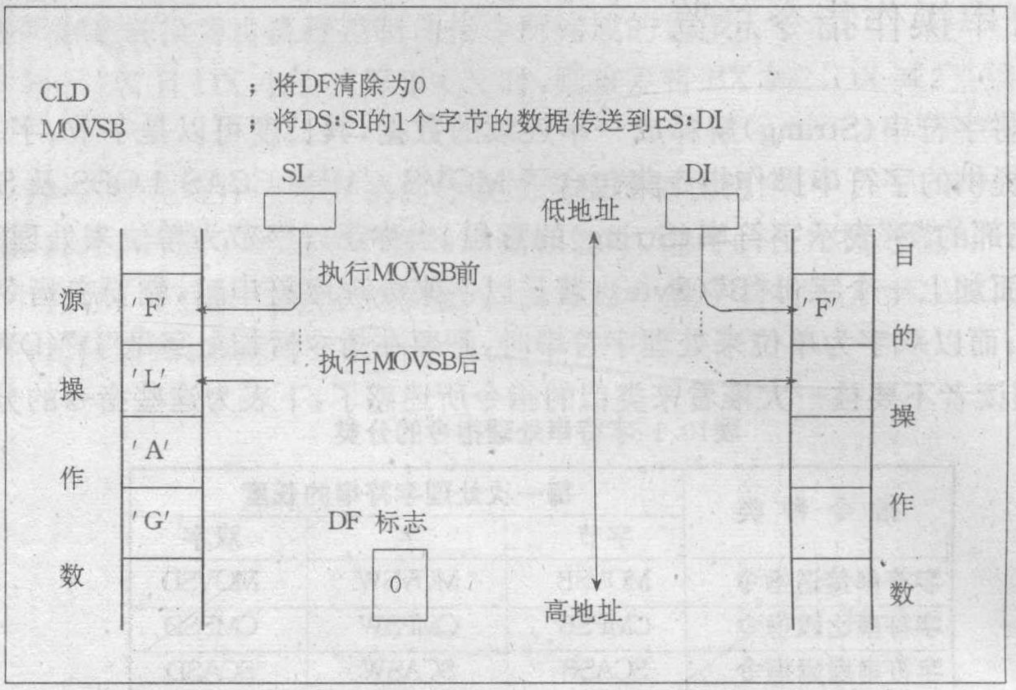


图10.1 MOVSB 指令执行前后SI与DI的指向

由于每次执行字符串传送指令后,SI 和 DI 都会自动指向下一个待处理的字符,因此我们可以配合 LOOP 循环指令或前缀指令 REP(稍后会说明),来传送1Byte(Word)以上的资料。现在让我们用一个 DEBUG 的例子来说明:

实例

```
-a
210D:0100  DB  'Source'            ;定义源操作串
210D:0106  DB  '      '            ;定义目的操作串(保留6个字节的空间)
210D:010C  MOV  SI,100             ;指向源操作串地址
210D:010F  MOV  DI,106             ;指向目的地址
210D:0112  MOV  CX,6               ;设定传送次数
210D:0115  MOVSB                   ;开始传送 <--
210D:0116  LOOP 115                ;传送6次  ---
210D:0118  INT  3
210D:0119                 SI、DI 分别指向源和目的操作数的偏移地址
-g=10C 115 <---- 由10C开始执行三条指令
DS、ES 已指向两操作数的段地址

AX=0000  BX=0000  CX=0006  DX=0000  SP=FFEE  BP=0000  SI=0100  DI=0106
DS=210D  ES=210D  SS=210D  CS=210D  IP=0115   NV UP EI PL NZ NA PO NC
210D:0115     A4          MOVSB
                        方向标志DF=0
-t <---- 传送一个字节
                 SI、DI已指向下一个数据的地址
AX=0000  BX=0000  CX=0006  DX=0000  SP=FFEE  BP=0000  SI=0101  DI=0107
DS=210D  ES=210D  SS=210D  CS=210D  IP=0116   NV UP EI PL NZ NA PO NC
210D:0116     E2FD        LOOP   0115
-g <---- 传送其余的5个数据
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0106  DI=010C
DS=210D  ES=210D  SS=210D  CS=210D  IP=0118   NV UP EI PL NZ NA PO NC
210D:0118     CC          INT    3
-d106L6 <----列出目的地址的内容
210D:0100                     53 6F-75 72 63 65                 Source
-
                        果然已传送过去了
```

◆ 注： 在DEBUG中，CS、DS、ES均指向同一段区，因此不必再另行设定DS及ES。

10.3 字符串比较指令

<table>
<tr><td>CMPSB、CMPSW、CMPSD</td><td>字串比较指令</td></tr>
<tr><td colspan="2">语　法： CMPSB ←—— 将DS:SI地址的一个字节数据和ES:DI的数据比较，再将SI及DI加1(或减1，视方向标志值而定)
CMPSW ←—— 将DS:SI地址的一个字的数据和ES:DI的数据比较，再将SI及DI加2(或减2，视方向标志值而定)
CMPSD ←—— 将DS:SI地址的一个双字的数据和ES:DI的数据比较，再将SI及DI加4(或减4，视方向标志值而定)
影响标志： C、P、Z、S、O
注:方向标志DF=1时，每比较一个字节(或字)后，SI和DI会被减去1(或2)</td></tr>
</table>

说明

CMPSB（CoMPare String Byte by byte）、CMPSW（CoMPare String Word by word）、CMPSD(CoMPare String Double word by double word)的功能是判断两个字符串是否相同。和字符串传送指令相似，字符串比较指令会比较 DS:SI 与 ES:DI 所指和两个字符串。每比较一个单位的数据(Byte、Word、或 Double Word)后，SI 与 DI 的变化方式和字符串传送指令完全相同。

字符串比较指令通常搭配 LOOPE、LOOPNE 循环指令或 REPE、REPNE 前缀指令来使用(稍后说明)。当以 LOOPE(REPE)来配合时，比较操作会持续到两字符串不相等或 CX 等于0为止。相反的，若使用 LOOPNE(REPNE)配合时，则比较操作会持续到两字符串相等或 CX 等于0为止。

字符串比较指令会影响 O、S、Z、A、P、C、等标志的值，但是只有 Z 标志是有用的，当 Z=1时，则代表两个字符串相等，而 Z=0时，则表示两字符串不相等。

实例

```
- a
210D:0100  DB  'FLAG PUBLISHING'       ;定义源操作串
210D:010F  DB  'FLAG RUN PC'           ;定义目的操作串
210D:011A  MOV  SI,100                 ;SI指向源操作串
210D:011D  MOV  DI,10F                 ;DI指向目的操作串
210D:0120  MOV  CX,0F                  ;设定要比较的次数
210D:0123  CMPSB                       ;开始比较
210D:0124  LOOPE  123                  ;比较到两字串中某一字符不等为止
210D:0126  CMPSB                       ;开始比较
210D:0127  LOOPNE  126                 ;比较到两字串中某一字符相等为止
210D:0129  INT  3
210D:012A
                             SI、DI分别指向源及目的地址
- g=11A 123 ←—— 由11A处开始执行三道指令
AX=0000 BX=0000 CX=000F DX=0000 SP=FFEE BP=0000 SI=0100 DI=010F
DS=210D ES=210D SS=210D CS=210D IP=0123   NV UP EI PL NZ NA PO NC
210D:0123   A6         CMPSB          DF=0
- t ←—— 比较第一个字符      SI、DI分别指向下一个字符地址
AX=0000 BX=0000 CX=000F DX=0000 SP=FFEE BP=0000 SI=0101 DI=0110
DS=210D ES=210D SS=210D CS=210D IP=0124   NV UP EI PL ZR NA PE NC
210D:0124   E1FD       LOOPZ  0123
- p ←—— 比较到某一字符不等为止    两字符相等，所以ZF=1
    比较到第5个字符'P'和'R'，SI、DI分别指向下一个字符地址
AX=0000 BX=0000 CX=0009 DX=0000 SP=FFEE BP=0000 SI=0106 DI=0115
DS=210D ES=210D SS=210D CS=210D IP=0126   NV UP EI NG NZ AC PO CY
210D:0126   A6         CMPSB
- t ←—— 比较第六个字符    两字符不等，所以结束循环，ZF=0
AX=0000 BX=0000 CX=0009 DX=0000 SP=FFEE BP=0000 SI=0107 DI=0116
DS=210D ES=210D SS=210D CS=210D IP=0127   NV UP EI PL ZR NA PE NC
210D:0127   E0FD       LOOPNZ  0126
- g ←—— 比较到某一字符相等为止    两字符相等，所以ZF=1
    比较到第6个字符'U'和'U'，SI、DI分别指向下一个字符地址
AX=0000 BX=0000 CX=0008 DX=0000 SP=FFEE BP=0000 SI=0107 DI=0116
DS=210D ES=210D SS=210D CS=210D IP=0129   NV UP EI PL ZR NA PE NC
210D:0129   CC         INT  3
-
                两字符相等，所以结束循环，ZF=1
```

10.4 字符串搜索指令

SCASB、SCASW	字符串搜索指令
语　法：SCASB ←—— 将AL的数据和ES:DI的一个字节数据做比较，再将DI加1(或减1，视方向标志值而定) SCASW ←—— 将AX的数据和ES:DI的一个字数据做比较，再将DI加2(或减2，视方向标志值而定) 影响标志：C、P、Z、S、O 注:方向标志DF=1时，每比较一个字节(或字)后，DI会被减去1(或2)	

说明

SCASB (SCAn String Byte by byte)、SCASW (SCAn string Word by word)与字符串比较指令类似,主要的不同在于,字符串搜索是以一个放在 AL(SCASB)或 AX(SCASW)的数据和 ES:DI 中的字符串相比较,比较完后仅 DI 会指向字符串的下一个字符。比较的结果可以由 Z 标志的值来判断:当两者相同时,Z 标志等于1,否则 Z 标志等于0。

同样地,欲搜索字符串中与 AL(AX)相同者,可使用 LOOPNE(REPNE)来控制,而欲搜索字符串中与 AL(AX)不同者,则可使用 LOOPE(REPE)来控制。

实例

```
-a
230F:0100  DB 'ABCDEFGHHHHHIJKL'        ;定义欲搜索的字符串
230F:0110  MOV  DI,100                  ;DI指向字符串起始地址
230D:0113  MOV AL,47                    ;将'G'存入AL
230F:0115  MOV CL,10                    ;设定搜索次数
230F:0117  SCASB                        ;开始搜索 <-----
230F:0118  LOOPNE  117                  ;搜索到字符串中发现'G'字符为止
230F:011A  MOV   AL,48                  ;将'H'存入AL
230F:011C  SCASB                        ;开始搜索 <-----
230F:011D  LOOPE  11C                   ;搜索到字符串中发现非'H'字符为止
230F:011F  INT 3
230F:0120
-g=110 117<-------由110处开始执行三行指令
          将'G'存入AL                          DI指向字符串起始地址
AX=0047  BX=0000  CX=0010  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0100
DS=230F  ES=230F  SS=230F  CS=230F  IP=0117  NV UP EI PL NZ NA PO NC
230F:0117    AE       SCASB
-t <---  搜索第一个字符                      DI指向字符串下一个地址
AX=0047  BX=0000  CX=0010  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0101
DS=230F  ES=230F  SS=230F  CS=230F  IP=0118  NV UP EI PL NZ NA PE NC
230F:0118   E0FD        LOOPNZ  0117
                                        两字符不相等
-p<---搜索到发现'G'字符为止            DI指向'G'字符的下一个地址
AX=0047  BX=0000  CX=0009  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0107
DS=230F  ES=230F  SS=230F  CS=230F  IP=011A  NV UP EI PL ZR NA PE NC
230F:011A   B048        MOV   AL, 48
-g11c  将'H'存入AL            两字符相等，跳出搜索循环，Z=1
AX=0048  BX=0000  CX=0009  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0107
DS=230F  ES=230F  SS=230F  CS=230F  IP=011C  NV UP EI PL ZR NA PE NC
230F:011C    AE       SCASB
-t <----- 继续搜索下一个字符
AX=0048  BX=0000  CX=0009  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0108
DS=230F  ES=230F  SS=230F  CS=230F  IP=011D  NV UP EI PL ZR NA PE NC
230F:011D   E1FD        LOOPZ  011C
                                             两字符相等
-p<---搜索到发现非'H'字符为止          DI指向'I'字符的下一个地址
AX=0048  BX=0000  CX=0003  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=010D
DS=230F  ES=230F  SS=230F  CS=230F  IP=011F  NV UP EI NG NZ AC PE CY
230F:011F    CC       INT    3
-                          两字符不相等，跳出搜索循环，Z=0
```

10.5 字符串的载入与储存指令

LODSB、LODSW、STOSB、STOSW	字符串载入或贮存指令
语　法： LODSB ←— 将DS:SI地址的一个字节数据载入到AL上，并将SI加1(或减1，视方向标志值而定) LODSW ←— 将DS:SI地址的一个字的数据载入到AX上，并将SI加2(或减2，视方向标志值而定) STOSB ←— 将AL的数据传送到ES:DI，并将DI加1(或减1，视方向标志值而定) STOSW ←— 将AX的数据传送到ES:DI，并将DI加2(或减2，视方向标志值而定) 影响标志： 无 注：方向标志DF=1时，传送一个字节(或字)后，SI和DI会被减去1(或2)	

说明

LODSB(LOaD one Byte from a String)、LODSW(LOaD one Word from a String)称为字符串载入指令，它们会将 DS:SI 的一个字节(字)的数据传送到 AL(AX)然后将 SI 指向字符串的下一个地址。

STOSB (STOre one Byte to a String)、STOSW (STOre one Word to a String)则称为而字符串储存指令，是将 AX 或 AL 寄存器内的数据存入 ES:DI 所指的地址上，并将 DI 指向字符串的下一个地址。以下为 LODSB 及 STOSB 的示意图。

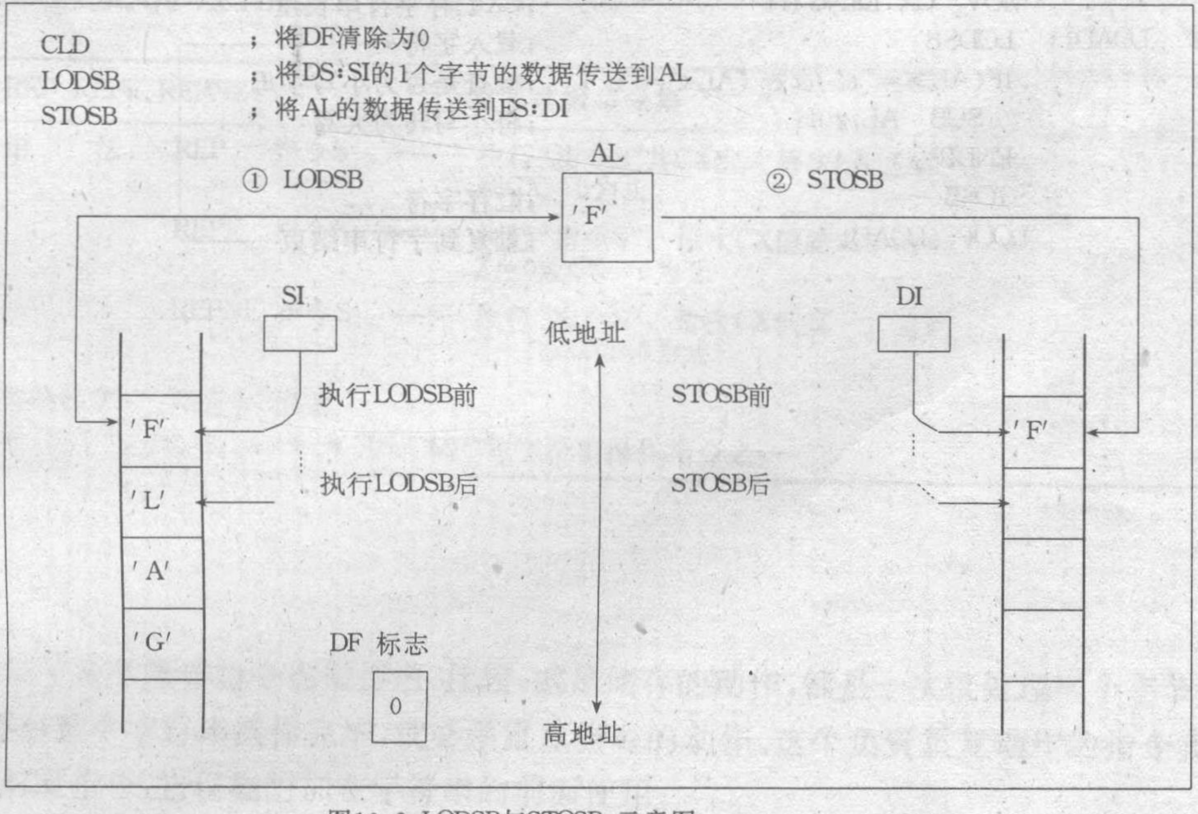


图10.2 LODSB与STOSB 示意图

由上图中我们可以看出，其实，将一个字符串载入指令与字符串储存指令合并就等于是一

个字符串传送指令，例如：

```
      CLD                  ;将DF清除为0
                           ┌ LODSB      ;将DS:SI的一字节传送到AL, SI加一
MOVSB——就等于 ——→ ┤
                           └ STOSB      ;将AL的数据传送到ES:DI, DI加一
```

此外，当我们要以一个固定的数据填满指定的存贮区时，可使用 LOOP 或 REP 配合 STOSB（STOSW）来完成，例如：

```
          ............
          MOV  AX,@DATA         ;
          MOV  ES,AX            ;将ES指向DATA段
          MOV  DI,00            ;将DI指向000h偏移量
          MOV  CX,20            ;设定要贮存的字符数目
          MOV  AL,'A'           ;设定要写入存贮器的数据
FILL:     STOSB                 ;写入数据 ←──────────────┐
          LOOP  FILL            ;由ES:0000地址开始写入20个'A' ─┘
          ............
          ............
```

由于 LODSB（LODSW）每次只将1 字节（字）的数据载入 AL（AX），我们可以将它与循环指令配合，并针对每一次载入的 AL（AX）值做特别处理，例如将字符串中的小写字母转成大写：

```
          PUSH  DS                        ;
          POP   ES                        ;将ES指向DS
          MOV   SI,OFFSET STRING          ;将SI指向字符串偏移量
          MOV   DI,SI                     ;将DI指向字符串偏移量
          MOV   CX,LENGTH                 ;CX贮存字符串长度
LOAD1:    LODSB                           ;载入字符 ←──────┐
          .IF(AL>='a')&&(AL<='z')         ;检查是否为小写字母  │
            SUB  AL,20H                   ;将小写转为大写      │
          .ENDIF                                              │
          STOSB                           ;贮存字符            │
          LOOP  LOAD1                     ;重复到字符串结束 ───┘
```

实例

```
—a

230F:0100  DB '0123456789'        ;保留10个字节空间
230F:010A  MOV  SI,100            ;SI指向字符串起始地址
230F:010D  LODSB                  ;将1个字节的数据由DS:SI处载入AL
230F:010E  MOV CX,0A              ;设定要贮存字符的个数
230F:0111  MOV DI,100             ;DI指向贮存字符串的起始地址
230F:0114  STOSB                  ;将AL存入ES:DI ←┐
230F:0115  LOOP 114               ;重复贮存10个字节 ┘
230F:0117  INT  3
230F:0118

—g=10A 10E←———— 由10A处开始执行到载入字符串为止
    ┌——AL由DS:SI处载入'0'这个字符        SI指向下一地址
    ↓                                                    ↓
AX=0030  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0101  DI=0000
DS=230F  ES=230F  SS=230F  CS=230F  IP=010E   NV UP EI PL NZ NA PO NC
230F:010E    B90A00       MOV    CX,000A
—g←—— 由ES:DI地址开始，存入10个'0'    DI指向字符串的第11个字节处 ——┐
                                                                   ↓
AX=0030  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0101  DI=010A
DS=230F  ES=230F  SS=230F  CS=230F  IP=0117   NV UP EI PL NZ NA PO NC
230F:0117  CC        INT       3
—d100LA   ←—显示字符串内容                      果然存入10个'0'
                                                              ↓
230F:0100  30 30 30 30 30 30 30 30—30 30                0000000000
—
```

10.6 重复前缀指令

REP、REPE、REPNE	重复前缀指令集
语　法：REP　指令S ←—— 执行"指令S"共CX次，每次CX值减一，直到CX=0为止 REPE　指令S ←—— 执行"指令S"，每次CX值减一，直到Z=0或CX=0为止 REPNE 指令S ←—— 执行"指令S"，每次CX值减一，直到Z=1或CX=0为止 影响标志：依指令S而定 注：指令S为MOVS、STOS、SCAS、CMPS等字符串操作指令之一	

说明

由于字符串操作指令在做传送、比较、或是储存的动作，都是一次仅处理一个字节(或字)，因此，要将整个字符串操作完毕，就必须重复指令的动作。这个负责重复动作的指令我们称之为重复前缀指令，它仅能附加在字符串的前面使用。

REP(REPeat)与LOOP指令十分类似，二者均使用CX来指明执行的次数。不同的地方在REP只能重复执行单一的字符串操作指令，而LOOP却可将多个指令纳入执行循环中，例如：

```
        REP   MOVSB              ;使用REP重复执行MOVSB指令CX次

  Fill:   MOVSB                  ;  ←——— 执行循环 ←————┐
          INC  BX                ;                       │
          LOOP Fill              ;执行MOVSB和INC BX这两行指令CX次
```

相对于 REP 与 LOOP 类似，REPE(REPeat if Equal)、REPNE(REPeat if Not Equal)则分别与 LOOPE、LOOPNE 相类似。REPE、REPNE 除了以 CX 为计数器之外，还会同时检查 Z 标志之值：REPE 会在 Z 标志为0或 CX＝0时停止循环。而 REPNE 则会在 Z＝1或 CX＝0时，停止循环。这些指令大多搭配字符串比较或搜索指令使用，例如：

```
REPE CMPSB ；比较至 CX＝0 或 DS:SI 与 ES:DI 中有字符不等时
REPNE SCASB；搜寻至 CX＝0 或 AL 的值与 ES:DI 中的字符相等为止
```

我们将重复前缀指令如何搭配各种字符串指令及指针寄存器的方法整理见表10.2。

实例

```
-a
61BA:0100 DB 'FLAG'        ;定义源字符串
61BA:0104 DB 'MASM'        ;定义目的字符串
61BA:0108 MOV SI,100       ;将SI指向源字符串地址
61BA:010B MOV DI,104       ;将DI指向目的字符串地址
61BA:010E MOV CX,4         ;设定传送的次数
61BA:0111 REP MOVSB        ;将DS:SI的4字节数据传送移到ES:DI
61BA:0113 INT 3
61BA:0114
-g=108 111 ←—— 由108开始执行三条指令
                              SI与DI分别指向源和目的地址
AX=0000  BX=0000  CX=0004  DX=0000  SP=FFEE  BP=0000  SI=0100  DI=0104
DS=61BA  ES=61BA  SS=61BA  CS=61BA  IP=0111   NV UP EI PL NZ NA PO NC
61BA:0111      F3         REPZ
61BA:0112      A4         MOVSB
-t                     SI与DI分别指向源和目的字符串的下一字符地址
 每执行一次CX会被减一
AX=0000  BX=0000  CX=0003  DX=0000  SP=FFEE  BP=0000  SI=0101  DI=0105
DS=61BA  ES=61BA  SS=61BA  CS=61BA  IP=0111   NV UP EI PL NZ NA PO NC
61BA:0111      F3         REPZ
61BA:0112      A4         MOVSB
-g                              执行完后CX=0
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0104  DI=0108
DS=61BA  ES=61BA  SS=61BA  CS=61BA  IP=0113   NV UP EI PL NZ NA PO NC
61BA:0113      CC         INT  3
-d104L4 ←——— 看看目的字符串内容
61BA:0100                    46 4C 41 47                        FLAG
-
                    果然已完成传送字符串操作 ——————————————————↑
```

表10.2 字符串指令可搭配使用的前缀与指针寄存器

指令	前　缀	指针寄存器
MOVS	REP	DS:SI、ES:DI
CMPS	REPE/REPNE	DS:SI、ES:DI
STOS	REP	ES:DI
SCAS	REPE/REPNE	ES:DI
LODS	无	DS:SI

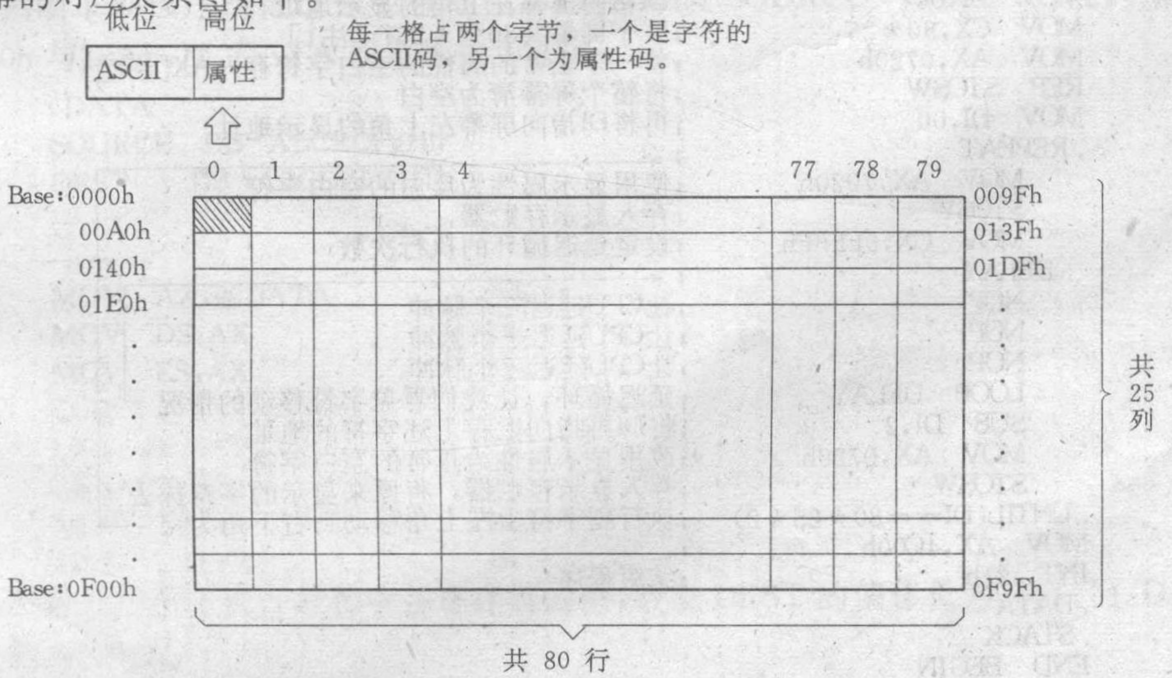
10.7 程序举例

本实例会将一个白色的方块由屏幕的左上角移动到右下角，为了便于解说，首先简单介绍在屏幕中快速显示文字的方法。

10.7.1 文字显示原理简述

PC 的显示状态可分为文本模式与图形模式两种，其中文本模式仅能显示文字，而图形模式则可以显示文字及图形数据。我们一般在 DOS 下均是使用文本模式，这种模式有一种快速显示文字的方法（不须通过 DOS 或 BIOS 调用），以下我们将简单介绍这个方法：

在文本模式下，屏幕上所显示的文本数据实际是储存在显示卡上的存贮器中，我们称之为 Video RAM。Hercules 卡（单色卡）是以 B000:0000－B000:0FFF 共4K 的地址区间来存放要显示的文本数据。而 VGA 卡（Video Graphic Adapter，彩色卡）则是使用 B800:0000－B800:0FFF 的4k 存贮器。因此，只要 ASCII 码存入 Video RAM，则对应的荧幕上便会显示该字符，存贮器与屏幕的对应关系图如下。



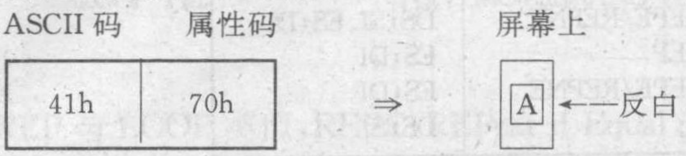
◆ 注1：　当显示卡为Hercules时，Base＝B000h，若为VGA卡时，则Base＝B800h。

◆ 注2：　在文本模式下仅用到80(行)×25(列)＝4000字节(0～0F9Fh)，由0FA0h～0FFFh的存贮器并没有用到。

图10.3　文本模式的存贮器与显示屏幕对应图

上图中屏幕的一个字符对应到 Video RAM 内的2个字节，第一个字节存放该字符的

ASCII 码值，第2个字节则存放其属性码(Attribute)。所谓属性码就是用来指定该字符的显示颜色，诸如反白、闪烁、蓝底黄字等等，以本实例中所用到的黑底白字颜色而言，其属性码为07h，而反白的属性码则为70h。例如：



为加深读者的印象，以下我们用一个 DEBUG 实例来说明：

```
A [A] ←——显示结果
C:\>DEBUG
-e B800:0000 41 07 41 70
  ↑      ↖——屏幕第0行的地址
  └—单色显示卡使用者请改为 B000
```

10.7.2 MASM 程序实例

程序 PRG10-1.ASM

```
          .8086
          .MODEL  SMALL
SCRSEG    EQU  0B800h             ;VGA卡文本模式的Video RAM段地址
                                  ;若是单色卡使用者请将0B800h改为0B000h
          .CODE
BEGIN:
          MOV  AX,@DATA
          MOV  DS,AX

          MOV  AX,SCRSEG          ;将ES指向Video RAM的段地址
          MOV  ES,AX              ;
          MOV  DI,00              ;DI指向屏幕左上角的显示地址
          MOV  CX,80*25           ;整个屏幕共80列*25行[注1]
          MOV  AX,0720h           ;将一个正常的属性的空白字符存入AX[注2]
          REP  STOSW              ;将整个屏幕清为空白
          MOV  DI,00              ;再将DI指向屏幕左上角的显示地址
          .REPEAT                 ;
            MOV  AX,7020h         ;使用显示属性为反白的空白字符
            STOSW                 ;存入显示存贮器
            MOV  CX,0FFFFh        ;设定延迟循环的执行次数
          DELAY:                  ;
            NOP                   ;让CPU延迟三个脉冲
            NOP                   ;让CPU延迟三个脉冲
            NOP                   ;让CPU延迟三个脉冲
            LOOP  DELAY           ;延迟循环，让我们看见字符移动的情况
            SUB  DI,2             ;将DI再指向贮存上述字符的地址
            MOV  AX,0720h         ;改用显示属性为正常的空白字符
            STOSW                 ;存入显示存贮器，将原来显示的字符抹去
          .UNTIL(DI==80*25*2)     ;执行到字符由左上角移动到右下角为止
          MOV  AX,4C00h           ;
          INT  21h                ;结束程序
          .DATA
          .STACK
          END  BEGIN
```

◆ 注1：MASM允许在常数值中使用四则运算，例如 MOV AX，(100-5)*4
注2：0720h的低位20h为ASCII码，高位07h则为属性码(黑底白字)。

程序说明

1)程序首先将一个正常属性的空白字符填满整个4000 Bytes 的 Video RAM,因此屏幕会被清为空白。

2)在处理移动方块的循环里,我们先将一个反白属性的空白字符(会显示出白色方块)填入存贮器内,然后执行由 NOP 所构成的延迟循环,让我们能看清这个方块,然后再用一个正常属性的空白字符存入同一地址,将该方块抹掉。

3)当 DI 指向下一个显示地址后,继续重复<画方块－清除方块>的步骤,直到 DI 指向屏幕右下角的显示地址才结束。

本章习题

1. 请问字符串操作指令可分为哪几类?
2. 请问 SI、DI 与字符串操作指令有何关系?
3. 请问 D 方向标志有何用途?
4. 请说明 MOVSB、MOVSW、CMPSB、CMPSW、SCASB、SCASW、LODSB、LODSW、STOSB、STOSW、REP、REPE、REPNE 指令的格式与功能?
5. 请判断下列程式执行完后的 SI 与 DI 值

```
CLD
MOV  DI,100
MOV  SI,200
MOV  CX,20
REP  MOVSW
```

6. 请判断下列程序执行过程中各寄存器的值,假设 SOURCE 与 DEST 的段内偏移量分别为 100h 与10Ah,而.DATA 段地址则为2000h。

```
.DATA
SOURCE  DB "ABCEDFgHij"
DEST    DB "ABCDEFGHIJ"
-----
-----
MOV  AX,@DATA              AX = ?
MOV  DS,AX                 DS = ?
MOV  ES,AX                 ES = ?
CLD
LEA  DI,DEST               DI = ?
LEA  SI,200                SI = ?
MOV  CX,20                 CX = ?
REPE  CMPSB                SI = ?, DI = ?
```

7. 请判断下列程序执行过程中各寄存器的值,假设 DEST 的偏移量为100h,而.DATA 的段地址则为2000h。

```
.DATA
DEST    DB "AABBCDEEFFGHIJ"
------
--------
MOV  AX,@DATA              AX = ?
MOV  DS,AX                 DS = ?
MOV  ES,AX                 ES = ?
```

```
CLD
MOV   AL,'F'                    AL = ?
LEA   DI,DEST                   DI = ?
MOV   CX,20                     CX = ?
REPNR   SCASB                   SI = ?, DI = ?
```

8. 请判断下列程式执行完毕后，DEST 的内容为何？

```
.DATA
    SOURCE      DB "FLAG PUBLISHING"
    DEST        DB 15 DUP('*')
    ————
    ——————————
    MOV   AX,@DATA
    MOV   DS,AX
    MOV   ES,AX
    CLD
    LEA   DI,DEST
    LEA   SI,SOURCE
    MOV   CX,6
FILL:
    LODSW
    STOSB
    LOOP  FILL
```

9. 请比较 REP、REPE、REPNE 与 LOOP、LOOPE、LOOPNE 之间的异同？

10. 请写一个程序将荧幕填满反白的 '@' 字符。

11. 请写一个程序以 'A' 字符在荧幕上排出一个 FLAG 的字样，如下图。

```
AAAAAAAA  AA         AA       AAAAA
AA        AA        A   A    A     A
AAAAAA    AA       A    A   A
AA        AA       AAAAAA   A    AAAAA
AA        AA       A    A    A     A
AA        AAAAAAA  A    A     A AA A
```

12. 请改写 PRG10-1. ASM 白色方块移动至荧幕右下角后，再以反方向移回左上角，如此周而复始，直到按下任意按键为止。

13. 请写一个程序，可以将屏幕上半部的数据(0～9行)拷贝到下半部(10～19行)。

14. 请写一个程序，可以将屏幕上的大写字母全部改为小写。

第四篇　高级应用

第十一章　子程序的使用

MASM 虽然只是一个低级的汇编语言编译器，却和高级语言一样，也提供了子程序的调用功能。本章便是针对子程序做一详细的探讨，内容包括：

□定义完整的子程序

□近程调用与远程调用

□内部调用与外部调用

□程序库的使用

11.1　如何定义子程序

在第7章介绍 CALL、RET 指令时，我们曾经说明了子程序（Procedure）及模块化（Modulize）的观念，相信读者应该还记得。子程序调用及返回方式如下。

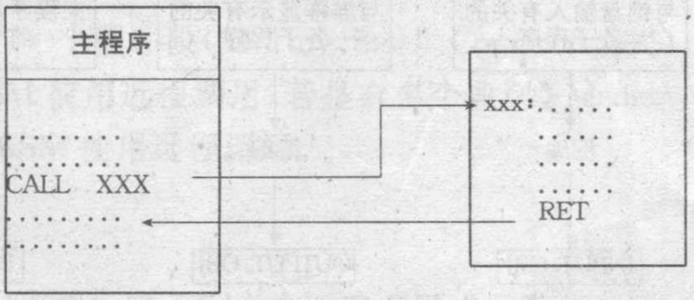


图11.1 调用子程序的执行流程

然而，如果一个程序中有很多的子程序，那么用程序标号来定义子程序将会显得很凌乱，而且所有的子程序都必须放在同一个程序当中，不仅不易阅读，也不易维护或再使用于别的程序中。

11.1.1　PROC 与 ENDP

有鉴于此，MASM 特别提供了定义子程序的伪指令：PROC 及 ENDP，可用来定义独立而自成体系的子程序。

```
子程序名称   PROC
             ....
             ....      } 子程序的内容
             ....
子程序名称   ENDP
```

PROC 与 ENDP（END of Procedure）是用来定义子程序的开始与结束，我们必须把子程序名称放在 PROC 的前头。子程序名最长32个字符，大小写均可。例如我们将 PRG8-1. ASM 中的两个子程序加以改写。

```
;名为InputStr的子程序
InputStr    PROC
            MOV DX,OFFSET BUF          ;DX指向字符串开头
            MOV AH,0Ah
            INT 21h                    ;执行DOS调用来读取字符串
            RET
InputStr    ENDP

;名为PrintStr的子程序
PrintStr    PROC
            MOV BUF,13                 ;将BUF最前面的2字节
            MOV BUF+1,10               ;  换成CR及LF
            MOV DX,OFFSET BUF          ;DX指向字符串开头
            MOV AH,09h
            INT 21h                    ;执行DOS调用来显示字符串
            RET
PrintStr    ENDP
```

11.1.2　子程序的汇编与连接

由 PROC 及 ENDP 定义的各个子程序可以存放在不同的程序文件中，分别汇编后再用 LINK 连接成一个可执行文件，例如：

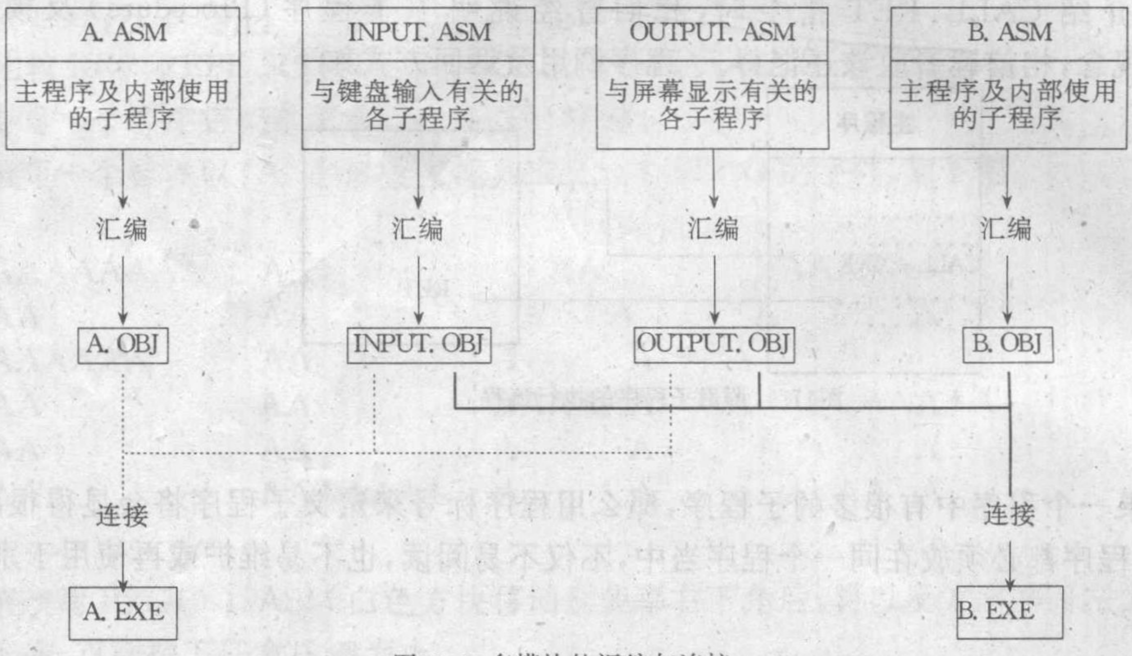


图11.2　多模块的汇编与连接

◆ 注：　图中A. EXE是A. OBJ、INPUT. OBJ、及OUTPUT. OBJ三者连接产生的。而B. EXE是B. OBJ、INPUT. OBJ、及OUTPUT. OBJ三者连接产生的。

上例中的 INPUT. ASM 及 OUTPUT. ASM 可看作共用的模块，我们只要编写一次后，便可以重复地使用于不同的应用程序中(如上例的 A、B 程序)。注意！当我们连接不同的程序文件时，它们的存贮器模式必须相同才行，否则会产生错误。

11.2　近程调用和远程调用

当我们用 CALL 来调用子程序时，如果该子程序是处在同一个段区中(即段地址相同)，则 CPU 会先把下一行指令的 IP 值压入堆栈，然后直接更改 IP 为子程序的段内偏移量而跳到

子程序去执行。像这种同一段内的调用我们称为“近程调用”(NearCall)。

若是跨段的调用，则 CPU 会先将 CS:IP 值压入推栈，然后再更改 CS:IP 子程序的地址而跳到子程序去执行。像这种不同段间的调用(因为 CS 段值也改了)我们称为“远程调用”(Far Call)。

当子程序以 RET 返回时，则会将堆栈中的值依近程或远程调用，而弹回 IP 或 CS:IP 中，以返回到原程序的下一行继续执行。

11.2.1 缺省的调用方式

在缺省的情况下，MASM 会以我们设定的存贮器模式来决定要使用近程或远程调用。

表11.1 存贮器模式与缺省子程序调用方式的对照

存贮器模式	程序段	子程序调用
Tiny (迷你)	一个	Near
Small (小型)	一个	Near
Medium (中型)	多个	Far
Compact (紧密)	一个	Near
Large (大型)	多个	Far

很明显的，如果只有一个程序段(如 Tiny、Small、Compact 模式)，那么调用子程序时 CS 的值不需改变，所以 MASM 使用近程调用。若是有多个段(如 Medium 或 Large 模式)，则 CS 的值会经常改变，所以 MASM 使用远程调用。

11.2.2 指定调用方式

MASM 也允许我们直接指定子程序为近程调用或远程调用，例如：

```
NCall  PROC Near
       ————
       ————
NCall  ENDP        ←—— 用Near指明此子程序要使用近程调用

FCall  PROC Far    ←—— 用Far指明此子程序要使用远程调用
       ————
       ————
FCall  ENDP
```

除非程序很复杂或有特殊用途，否则我们一般都使用 Small Model，而 SmallModel 会缺省使用 Near Call，因此不太需要使用 Near 或 Far 来指定调用方式。

11.3 内部调用与外部调用

在调用子程序时，若该子程序位于同一个程序文件中，则称为“内部调用”，若位于不同的程序文件中，则称为“外部调用”，请参考下图。

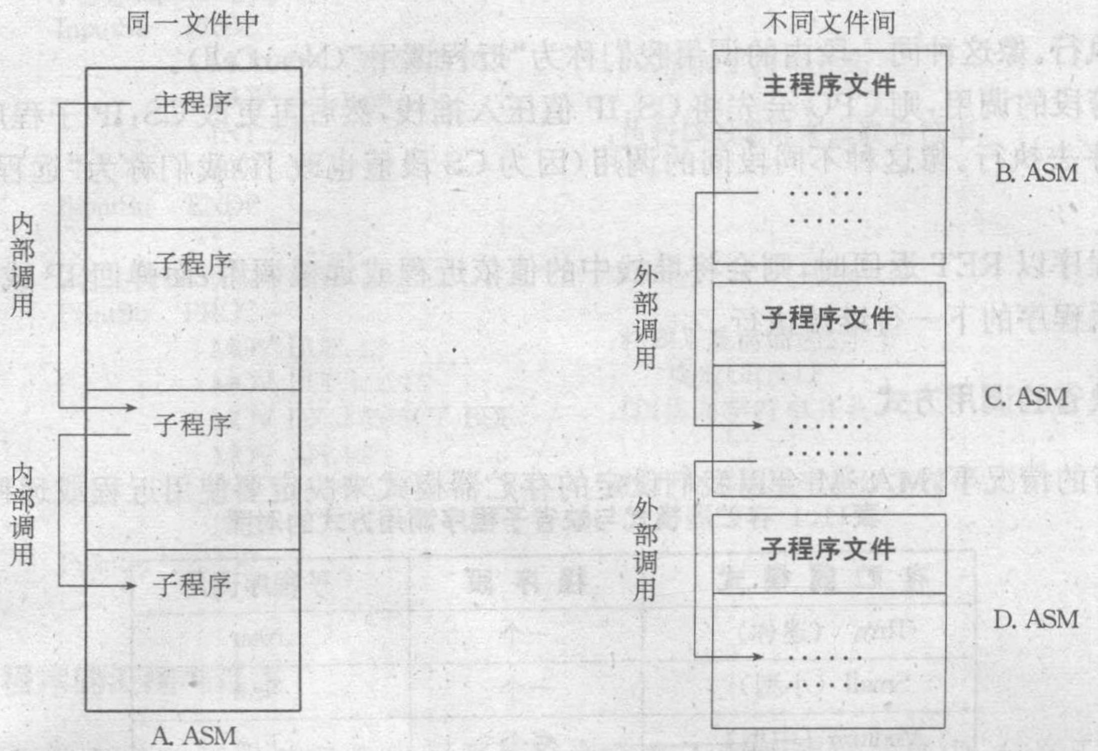


图11.3 内部调用与外部调用的情形

11.3.1 内部调用

同一个程序文件内的子程序调用就称为内部调用，下面是一个实例。

程序 PRG11-1.ASM

```
            .8086
            .MODEL SMALL

            .DATA
MSG         DB  "Hello Procedure! $"

            .CODE
BEGIN:  MOV   AX,@DATA                  ;DX指向字符串地址
        MOV   DS,AX                     ;调用PutStr子程序
        MOV   DX,OFFSET MSG
        CALL  PutStr
        MOV   AH,4Ch
        INT   21h

;------------------------------------
;子程序PutStr：将DS:DX所指的字符串显示于屏幕
;------------------------------------
PutStr  PROC
        MOV   AH,09h
        INT   21h
        RET
PutStr  ENDP

        .STACK
        END BEGIN
```

程序本身分为两个部分：主程序与子程序。主程序在调用子程序 PutStr 后，会将控制权传送给 PutStr，其执行流程如下图所示。

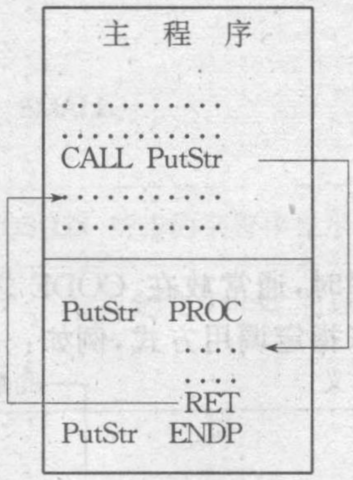


图 11.4

这个程序经汇编及连接后,执行结果如下:

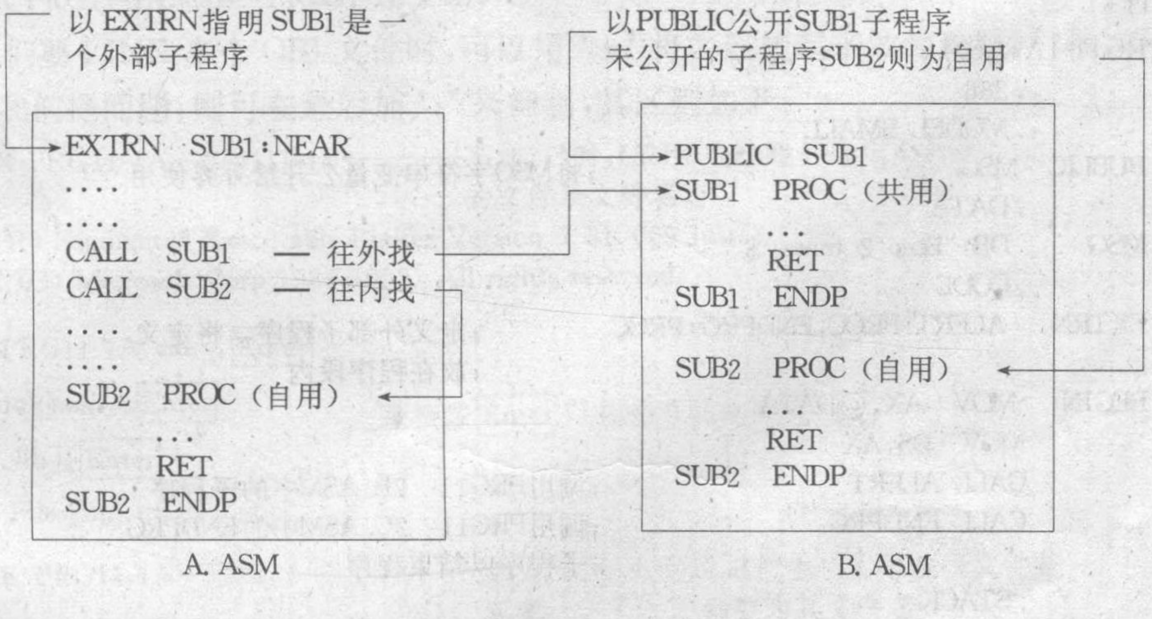
Hello Procedure!

11.3.2 外部调用

外部调用比内部调用多了二道手续,那就是:

1)若子程序是要提供给其他文件中的程序调用,则必须以 PUBLIC 定义。

2)若要调用的子程序是在其他文件中,则该子程序名必须先以 EXTRN 定义,并指明是近程调用、远程调用、或使用缺省方式。



读者应该不难看出,这样做的目的就是要避免混乱!也就是说,凡是未经 PUBLIC 定义的子程序都只限于该文件内部使用,如此才不用担心文件内自用的子程序名称,会与其他外部子程序因同名而造成错误。

同理,EXTRN 则告诉 MASM 要到其他文件中去寻找被调用的子程序,否则 MASM 只会在目前文件中寻找此子程序。当然,我们也必须指明外部子程序是近程调用还是远程调用,因为 masm 在汇编时,并不知道其他程序文件中子程序的定义情形。

PUBLIC 定义通常放在子程序本身的上一行,例如:

```
PUBLIC      ButStr
PutStr      PROC
            ------
            ------
PutStr      ENDP
```

用 EXTRN 来定义外部子程序时，通常放在.CODE 之后(程序段内)，但也可以在任何段之外，并在子程序名称之后用'：'来指定调用方式，例如：

```
在程序段内定义                                    放在所有段外定义

            .CODE                                  EXTRN   PutStr:Near
EXTRN   PutStr:Near                                .DATA
            ......                                 ......
            ......                                 ......
            ......        使用近程调用             .CODE
```

接于'：'之后的值可为 NEAR、FAR、或 PROC，其中 PROC 表示要使用缺省的方式(根据当时的存贮器模式而定)。

接着，我们将 PRG11-1.ASM 中的主程序与子程序分别独立出来，另外储存成两个独立的程序文件。

程序 PRG11-1A.ASM

```
            .286
            .MODEL SMALL
PUBLIC  MSG                               ;将MSG字符串变量公开给外界使用
            .DATA
MSG         DB "Beep 2 times! $ "
            .CODE
EXTRN   ALERT:PROC,ENDPRG:PROC            ;定义外部子程序，将定义
                                          ;放在程序段内
BEGIN:  MOV   AX,@DATA
        MOV   DS,AX
        CALL  ALERT                       ;调用PRG11- 2B.ASM中的子程序
        CALL  ENDPRG                      ;调用PRG11- 2C.ASM中的ENDPRG
                                          ;子程序以结束程序
        .STACK
        END   BEGIN                       ;程序结束
```

程序 PRG11-1B. ASM

```
            .8086
            .MODEL    SMALL
            .CODE
;-----------------------------------
;子程序 PutStr :将 DS:DX 所指的字符串显示于屏幕
;-----------------------------------
PUBLIC  PutStr                ; 定义 PutStr 可以被外部调用
PutStr  PROC
            MOV  AH,09h
            INT  21h
            RET
PutStr  ENDP
            END
```

请在 DOS 下执行:

C:\>ML PRG11-1A. ASM PRG11-1B. ASM

执行完后则会产一个 PRG11-1A. EXE(ML 会以第一个文件名参数为缺省的执行文件名),执行此程序的结果和 PRG11-0相同。当然,我们也可以将两个. ASM 程序分开汇编成. OBJ 文件后,再用 LINK 联结在一起。

C:\>MASM PRG11-1A. ASM

C:\>MASM PRG11-1B. ASM

C:\>LINK PRG11-1A. OBJ+PRG11-1B. OBJ;

当我们要 LINK 多个 OBJ 文件时,可以用'+'将之连接起来做为参数。另外,如果不想让 LINK 向我们提问题,则可在最后加';'来抑制,其区别如下:

```
C:\>LINK PRG11-1A+PRG11-1B←———不加';'时,LINK 会让我们确定所欲产生
                                的文件及文件名。
Microsoft (R) Segmented Executable Linker Version 5.31.009 Jul 13 1992
Copyringht (C) Microsoft Corp 1984-1992.  All rights reserved.

Run File [PRG11-1A.exe]: Enter          ┐
List File [nul.map]: Enter              │直接按 Enter 以接受系统缺省值
Libraries [.lib]: Enter                 ┘
Definitions File [nul.def]: Enter

C:\>LINK PRG11-1A+PRG11-1B;←———加';'以后,LINK 只产生一个可执行文件并
                                以第一个文件名参数为其文件名
Microsoft (R) Segmented Executable Linker Version 5.31.009 Jul 13 1992
Copyright (C) Microsoft Corp 1984-1992.  Ali rights reserved.

C:\>
```

11.4 PUBLIC 与 EXTRN 的使用

除了子程序外,PUBLIC 与 EXTRN 也可以使用在汇编语言的任何代名上。所谓代名,包括了子程序名、程序标号、变量名、及常数名。因此,不仅子程序可以作外部连接,程序标号、变量、常数也可以作外部连接。例如在 B. ASM 中可以使用 A. ASM 中定义的变量 Price。

```
        .MODEL SMALL
PUBLIC  Price  ←—— 标明给外
        .....       部使用
        .....
        .DATA
Price   DB  100
        .....
        .....
```

A. ASM

```
EXTRN Price: BYTE  ←— 使用
.CODE                 外部
.....                 变量
MOV AL, Price
.....
.....
.....
```

B. ASM

11.4.1　PUBLIC

PUBLIC 定义是用来将代名定义成"共用的"，其格式如下：

PUBLIC　代名[,代名][,代名]...

注：[]表示其内的符号或名称可以省略。

在 PUBLIC 后面可以有一个到多个代名，不过至少必须有一个。各代名间必须以逗号分开。

11.4.2　EXTRN

EXTRN 的使用格式如下：

EXTRN　代名:属性[,代名:属性]...

做 EXTRN 定义时，必须同时定义代名的属性。

□与子程序名或程序标号有关的属性包括：

1)NEAR：表示此子程序是使用近程调用。

2)FAR：表示此子程序是使用远程调用。

3)PROC：表示要 MASM 存贮器模式来决定调用方式。

□与变量名有关的属性包括：

1)BYTE：以字节为单位来存取数据。

2)WORD：以字为单位来存取数据。

3)DWORD：以双字为单位来存取数据。

4)FWORD：以6字节为单位来存取数据。

5)QWORD：以8字节为单位来存取数据。

6)TBYTE：以10字节为单位来存取数据。

□常数名的属性则一律用 ABS。

在使用 EXTERN 时必须特别注意定义的位置，一般而言，当所定义的是数据变量时，则可将定义放在数据段内，而定义子程序或程序标号时，则可放在程序段内，若是在 EXTERN 后同时定义变量、子程序等不同型别的代名时，则必须将定义放在所有的段之外。EXTERN 的定义位置整理如下。

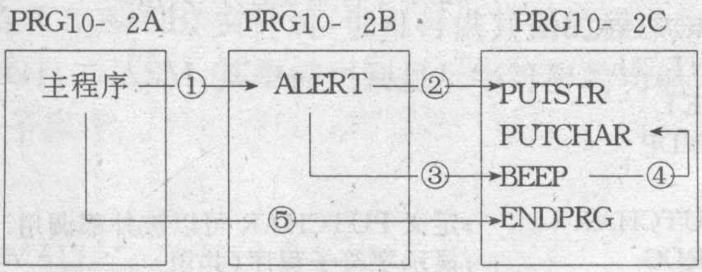
表11.2 EXTRN 定义放置位置

定义的类别	EXTRN 定义放置位置		
	数据段内	程序段内	所有段之外
数据变量	○	×	○
子程序或程序标号	×	○	○
数据变量、子程序等不同类别	×	×	○

11.5 程序举例

实 例

以下实例的功能相当简单，就是显示出一个字符然后发出两个哔声。比较复杂的地方在于它是由三个程序文件组合而成，我们将其中的子程序与执行次序整理如下。



在主程序中会调用 ALERT，而 ALERT 则先利用 PUTSTR 将 PRG11-2A 中定义的 MSG 字符串显示出来，然后再连续调用 BEEP 二次来发出声音。在 BEEP 中则是调用 PUTCHAR 来将 ASCII 7显示出来，由于 DOS 的显示字符功能会把 ASCII 7视为发声指令，所以就发出哔声了。

程序 PRG11-2A. ASM

```
        .286
        .MODEL SMALL
PUBLIC  MSG                          ;将MSG字符串变量公开给外界使用
        .DATA
MSG     DB "Beep 2 times! $ "
        .CODE
EXTRN   ALERT:PROC,ENDPRG:PROC       ;定义外部子程序，将定义
                                     ;放在程序段内
BEGIN:  MOV  AX,@DATA
        MOV  DS,AX
        CALL ALERT                   ;调用PRG11- 2B.ASM中的子程序
        CALL ENDPRG                  ;调用PRG11- 2C.ASM中的ENDPRG
                                     ;子程序以结束程序
        .STACK
        END  BEGIN                   ;程序结束
```

程序 PRG11-2B. ASM

```
        .286
        .MODEL SMALL
EXTRN   MSG:BYTE,BEEP:PROC,PUTSTR:PROC   ;定义要使用的外部变量及
                                         ;子程序，定义必须放在所有段之外
        .CODE
        PUBLIC  ALERT
ALERT   PROC
        MOV  DX,OFFSET MSG               ;DX指向MSG外部变量
        CALL PUTSTR                      ;调用PUTSTR外部子程序
        CALL BEEP                        ;调用BEEP外部子程序
        CALL BEEP                        ;
        RET                              ;返回主程序
ALERT   ENDP
        END
```

程序 PRG11-2C. ASM

```
                .286
                MODEL SMALL
                .CODE
        PUBLIC  PUTSTR          ;定义 PUTSTR 可以被外部调用
        PUTSTR  PROC            ;显示字符串子程序(共用)
                MOV AH,09h
                INT 21h
                RET
        PUTSTR  ENDP

        PUBLIC  PUTCHAR         ;定义 PUTCHAR 可以被外部调用
        PUTCHARPROC             ;显示字符子程序(共用)
                MOV AH,02h
                INT 21h
                RET
        PUTCHARENDP

        PUBLIC  BEEP            ;定义 BEEP 可以被外部调用
        BEEP    PROC            ;会发出 BEEP 声的子程序(共用)
                MOV DL,7
                CALL PUTCHAR
                                ;调用 PUTCHAR
                RET
        BEEP    ENDP

        PUBLIC  ENDPRG          ;定义 ENDPRG 可以被外部调用
        ENDPRG  PROC            ;结束程序的子程序(共用)
                MOV AX,4C00h
                INT 21h
                RET
        ENDPRG  ENDP
                END
```

执行结果

```
C:\>ML PRG11-2?. ASM←———ML 允许用通配符来指定文件名
Microsoft (R) Macro Assembler Version 6.11
```

```
Copyright (c) Microsoft Corp 1981-1993. All rights reserved.
    Assembling:PRG11-2A.ASM
    Assembling:PRG11-2B.ASM
    Assembling:PRG11-2C.ASM

Microsoft (R) Segmented Executable Linker Version 5.31.009 Jul 13 1992
Copyright (C) Microsoft Corp 1984-1992. All rights reserved.

Object Modules [.obj]:PRG11-2A.obj+
Object Modules [.obj]:"PRG11-2B.obj"+
Object Modules [.obj]:"PRG11-2C.obj"
Run File [PRG11-2A.exe]:"PRG11-2A.exe"
List File [nul.map]:NUL
Libraries [.lib]:
Definitions File [nul.def]:

C:\>PRG11-2A [Enter]
Beep 2 times!←———字符串显示后还会有两个哔声
```

11.6 子程序的再利用（Reuse）

编好的子程序模块在汇编成 OBJ 文件后，也可再供其他的程序来连接使用。例如我们要编写另外一个程序叫 PRG11-3.ASM，其中需用到显示字符串的功能，那么就可利用 PRG11-2C.ASM 中的 PUTSTR 子程序。

程序 PRG11-4.ASM

```
        .286
        MODEL SMALL
        .DATA
EXTRN   PUTSTR:PROC  ;定义要使用外部子程序 PUTSTR
BEGIN:  MOV AX,@DATA
        MOV DS,AX
        MOV DX,OFFSET MSG
        CALL PUTSTR  ;调用 PRG11C.ASM 中的子程序
        MOV AX,4C00h ;结束程序
        INT 21h      ;
        .STACK
        END BEGIN    ;结束程序
```

接著，我们看看如何汇编、连接 PRG11-3.ASM 及 PRG11-2C.OBJ：(PRG11-2C.ASM 在上一节已汇编过，所以可直接使用其 OBJ 文件)

```
C:\>ML PRG11-3.ASM PRG11-2C.OBJ ←———可直接指明要连接的 OBJ 文件
Microsoft (R) Macro Assembler Version 6.11
Copyright (c) Microsoft Corp 1981-1993. All rights reserved.

    Assembling:PRG11-3.ASM
Microsoft (R) Segmented Executable Linker Version 5.31.009 Jul 13 1992
Copyright (C) Microsoft Corp 1984-1992. All rights reserved.

Object Modules [.obj]:PRG11-3.obj+
Object Modules [.obj]:"PRG11-2C.obj"
Run File [PRG11-3.exe]:"PRG11-3.exe"
```

```
List File [nul.map]:NUL
Libraries [.lib]:
Definitions File [nul.def]:

C:\>PRG11-3 Enter ←———执行看看
OBJ can be reused!←———执行结果
```

ML 的参数在使用时非常有弹性，不仅允许我们使用通配符来指定文件名，而且也会自动以扩展文件名来判别文件的种类。包括.ASM、.OBJ、LIB、.DEF 四种，其中.DEF 是专供 Windows 上的应用程序使用，而.LIB 则会在下一节中介绍。ML 若是遇到.OBJ 或.LIB 文件，则就不会再汇编了，而是将它送给 LINK 进行连接。

当然，您也可以先用 MASM.EXE 将个别的程序文件汇编好，最后再以 LINK 来连接，不过使用 ML 还是比较方便一些。

11.7 程序库

由于子程序编好之后可以一再重复使用，所以当我们拥有的子程序愈来愈多时，编出来的主程序就愈具威力！不过每次 LINK 都要 PRG1＋PRG2＋PRG3＋…实在是太麻烦了，尤其这些模块并不是放在同一目录下时，LINK 往往会找不到模块而无法顺利操作。

程序库(Library)的目的，主要就是用来帮助我们管理那些已经编好，而且也汇编成 OBJ 文件的各个模块。所谓程序库，就是将已经汇编好的各种 OBJ 文件集合起来存成一个文件，并以.LIB 为其扩展文件名。

11.7.1 如何制作程序库

接下来，我们以 LIB.EXE 公用程序来将上一节汇编好的 PRG11B.OBJ 及 PRG11C.OBJ 集合为程序库，并以 UTIL.LIB 为程序库的文件名。

```
C:\>LIB  ←———以 LIB 公用程序来建立程序库

Microsoft (R) Library Manager Version 3.20.010
Copyright (c) Microsoft Corp 1983-1992. All rights reserved.
Library name:UTIL  ←———填入 UTIL 为程序库名(会自动加上.LIB)
Library does not exist. Create?(y/n)Y  ←———输入 Y 表示要建立新文件
Operations:+PRG11-2B+PRG11-2C  ←———以'+'表示要加入 OBJ 文件
List file:UTIL.LST←———将程序库的内容清单存于 UTIL.LST
```

LIB 首先会问我们程序库的名称，我们把程序库叫做 UTIL.LIB，由于它是一个新的程序库，所以 LIB 会问我们是否要建立，此时回答'Y'。接著 LIB 会问我们操作内容(OperaTIONS):，回答"＋PRG11-2B＋PRG11-2C"的意思是说要把 PRG11-2B.OBJ 和 PRG11-2C.OBJ 加到程序库中。以后我们还可以加入更多的程序模块。

最后我们回答"UTIL.LST"要求一份程序库的清单文件，其内容如下：

```
ALERT ........................ PRG11-2B          BEEP ............................ PRG11-2C
ENDPRG ....................... PRG11-2C          PUTCHAR ......................... PRG11-2C
PUTSTR ....................... PRG11-2C

PRG11-2B            Offset:00000010H Code and data size:dH ←──────┐
  ALERT←─── PRG11-2B 里的子程序            PRG11-2B 的大小(字节)──┘
                                           PRG11-2C 的大小(字节)──┐
PRG11-2C            Offset:000000d0H Code and data size:16H ←─────┘
  BEEP             ENDPRG             PUTCHAR             PUTSTR
  └──────────────────────────────────┬──────────────────────────────┘
                          PRG11-2C 里的子程序
```

在清单的前半部是将子程序名称来排列，并在每个子程序名的右方指明所属的 OBJ 文件。后半部则是将 OBJ 文件名来排列，并在每个文件名之下列出其内的所有子程序。

11.7.2 程序库的使用

一般我们只将具有 PUBLIC 性质的子程序模块放入程序库(因为只有公用性质的子程序才可以由我们的程序调用)。程序库的连接方式和子程序模块类似，请参考下图。

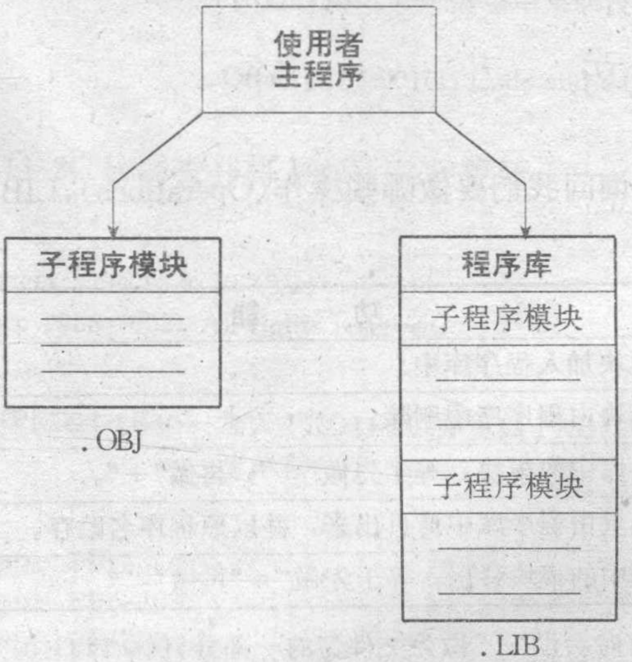


图11.5 子程序模块(OBJ文件)与程序库的比较

在使用程序库时，若我们是用 ML 做汇编连接，则只要将程序库名加进参数行即可。

```
C:\>ML PRG11-2A. ASM UTIL. LIB
Microsoft (R) Macro Assembler Version 6.11
Copyright (C) Microsoft Corp 1981-1993. All rights reserved.

  Assembling:PRG11-2A. ASM

Microsoft (R) Segmented Executable Linker Version 5.31.009 Jul 13 1992
Copyright (C) Microsoft Corp 1984-1992. All rights reserved.

Object Modules [.obj]:PRG11-2A. obj
Run File [PRG11-2A. exe]:"PRG11-2A. exe"
List File [nul. map]:NUL
```

Libraries [.lib]:"UTIL.LIB" ←——ML 会自动将之加入连接的程序库中
Definitions File [nul.def]:

若是使用 LINK 来连接汇编好的 OBJ 文件,则可在 Libraries 项目中填入程序库名称即可。

C:\>LINK PRG11-2A

Microsoft (R) Segmented Executable Linker Version 5.31.009 Jul 13 1992
Copyright (C) Microsoft Corp 1984-1992. All rights reserved.

Run File [PRG11-2A.exe]: Enter
List File [nul.map]: Enter
Libraries [.lib]: UTIL←——指定 UTIL.LIB
Definitions File [nul.def]: Enter

注意,LINK 在连接程序库时,只会把有使用到的 OBJ 模块连接到执行文件中,而不会把程序库中所有的 OBJ 文件都加进执行文件。举例来说,如果 TEST.LIB 中包含了 A、B、C 三个 OBJ 模块,而我们的程序只使用到 A.OBJ 中的 ABC 子程序,那么在连接时只有 A.OBJ 会被连接,其他的 B、C 模块则不会被连接。

另外,在连接时的最小单位是 OBJ 文件,例如 A.OBJ 有3个子程序,则虽然主程序只使用到 A.OBJ 中的一个子程序,但整个 A.OBJ 都会连接到执行文件中。

11.8 LIB 的使用技巧

每次在执行 LIB 时,它会询问我们要做哪些操作(Operations),LIB 共有5种不同的操作方式。

符号	功　能
+	将程序模块加入程序库中。
-	将程序模块由程序库中删除。
- +	调换程序库中的模块, 等于先做"- ", 再做"+"。
*	将程序模块由程序库中拷贝出来, 并以原程序名贮存。
- *	将程序库中的模块移出, 等于先做"*"再做"- "

◆ 注: 我们有时会以'-'做为文件名的一部分,例如PRG10-0.ASM, 但这样的文件名却无法使用在LIB中, 因会与操作符号'-'冲突。

另外 MASM 提供两个特殊符号:"&"和";"符号。如果欲键入的程序库太多太长,则可以键入"&"符号而由下一行接着继续输入。如果我们不想回答接下来的询问,可以键入";"号,LIB 便会省略其他的询问而开始执行。如果我们想中止 LIB 运行,则只要按下 Ctrl + C 键便可以了。

下面我们以 UTIL.LIB 为例来示范几个操作。

C:\>DIR PRG11-2B.OBJ
....
PRG11-2B.OBJ   168 05-22-95   2:46p
....

```
C:\>DEL PRG11-2B.OBJ←——删除 PRG11-2B.OBJ

C:\>LIB

Microsoft (R) Library Manager Version 3.20.010
Copyright (c) Microsoft Corp 1983-1992. All rights reserved.
Library name:UTIL
Operations: * PRG11-2B;←——将 PRG11-2B.OBJ 由程序库中拷贝出来
                      ↖——抑制接下来的询问

C:\>DIR PRG11-2B.OBJ
....
PRG11-2B.OBJ                168   05-22-95    4:05p←———和原来的一样
....                                               (除了建文件日期)
```

除了先执行 LIB 然后再回答问题外,我们也可以将要回答的问题以参数的方式一次传给 LIB(各参数间要以逗号分开)。

```
C:\>LIB UTIL-PRG11-2C,T;←———删除 PRG11-2C,并将清单存于 T 文件

Microsoft (R) Library Manager Version 3.20.010
Copyright (c) Microsoft Corp 1983-1992. All rights reserved.

C:\>TYPE T

ALERT ................................ PRG11-2B  ←—清单中果然没有 PRG11-2C 了

PRG11-2B                       Offset:00000010H Code and data size: dH
  ALERT
```

最后,我们将 PRG11-2C 放回去并将 PRG11-2B 更新。

```
C:\>LIB

Microsoft (R) Library Manager Version 3.20.010
Copyright (c) Microsoft Corp 1983-1992. All rights reserved.

Library name:UTIL
Operations:+PRG11-2C-+PRG11-2B←—加入 PRG11-2C 并更新 PRG11-2B
List file:T;                  ←—清单存入 T

C:\>TYPE  T
ALERT ..................... PRG11-2B        BEEP ....................... PRG11-2C
ENDPRG .................... PRG11-2C        PUTCHAR .................... PRG11-2C
PUTSTR .................... PRG11-2C

   ↙——PRG11-2C 果然又被加入了

PRG11-2C                 Offset:00000010H Code and data size: 16H
  BEEP                   ENDPRG                  PUTCHAR            PUTSTR

PRG11-2B                 Offset:000000d0H Code and data size: dH
  ALERT
```

如果我们想将 LIB 操作的结果存入另一个程序库,而不改变原来的程序库,则可以在 LIB 的第四个询问中输入新程序库的名称。下面的例子将 UTIL.LIB 中的 PRG11-2B 删除,然后把操作的结果存入 UTIL2.LIB,并将新程序库的清单存入 T 文件。

```
C:\>LIB

Microsoft (R) Library Manager Version 3.20.010
Copyright (c) Microsoft Corp 1983-1992. All rights reserved.
Library name:UTIL
```

Operations：-PRG11-2B←—将 PRG11-2B 由程序库中删除
List file：T
Output library：UTIL2←—将改动的结果存入另一个程序库

本章习题

1. 请解释下列词？

   子程序(Procedure)
   近程调用(Near Call)
   远程调用(Far Call)
   共用(Public)
   自用(Private)
   程序库(Library)

2. 请说明模块化及子程序的优点。
3. 请说明近程调用与远程调用的差异。
4. 请说明内部调用与外部调用的差异。
5. 请问 PROC 的属性参数 NEAR 和 FAR 各有何功能?若不指定则缺省值为何?
6. 请问 ASCLL 值为7的字符有何特殊用途?
7. 试说明 EXTRN 与 PUBLIC 两个伪指令的使用时机与格式?
8. 请问使用程序库有什么好处?要如何制作程序库呢?
9. 请简述 LIB 各项操作符号的功能，以及如何将程序库的内容清单列出。
10. 请将 PRG11-2B. ASM、PRG11-2C. ASM 中的每一个子程序都独立成一个文件，然后 分别汇编成 OBJ 文件后，再以 LIB. EXE 将它们制成程序库 PRG11. LIB，最后将 PRG11-2A 与 PRG11. LIB 连接成执行文件，并测试执行结果是否正确。
11. 请将上一题中，PRG11. LIB 的内容制成. LST 文件并显示出结果。
12. 当我们使用程序库做连接时，是否程序库中所有的模块都会被连接呢?另外，在被连接的模块中除了含有被调用的子程序外，是否可能包含没使用到的子程序呢?
13. 下面两个程序在连接时会发生那些错误?要如何修改?

A. ASM

```
            .MODEL SMALL
            .CODE
            ————
            CALL BB

PUBLIC      AA
    AA      PROC
            ———
            ———
    AA      ENDP
```

B. ASM

```
            .MODEL LARGE
            .CODE
PUBLIC      AA
    AA      PROC
            ———
    AA      ENDP

    BB      PROC
            ———
            ———
    BB      ENDP
```

14. 请将本书中所有介绍过的 DOS 中断调用功能编写成子程序，并存成 DOS. ASM，再编写一个主程序 TESTDOS. ASM 来测试其中的四个子程序是否能正确执行。
15. 请将上一题中的每一个子程序都独立建成一个文件，并以 DOS01、DIS02…为名，然后将这些程序成 DIS. LIB，再与 TESTDIS. ASM 连接成执行文件。
16. 请问前两题中所生成的执行文件哪一个比较小?为什么?

第十二章 宏(MACRO)的使用

MASM 的全名是 MACRO Assembler，正如其名，MASM 具有很强的宏(MACRO)能力，而本章就是要介绍 MACRO 的概念及使用，内容包括：

□MACRO 的原理与使用方法

□MACRO 与子程序的差异

□INCLUDE 伪指令的用途

□MACRO 的参数传递功能

□如何在宏内使用程序标号

12.1 何谓宏

我们知道，在调用子程序时，CPU 会先 PUSH IP(CS:IP)，而当子程序返回时，则还要 POP IP(CS:IP)。如果子程序的内容很短而且被调用的次数相当多，那么使用子程序虽然能增加可读性，但太多的 PUSH、POP 动作却会造成程序执行额外的负担！是否有其他变通之道呢？

以前曾介绍过 EQU 伪指令可用来设定常数值，例如以 CR 来代表数值0Dh。

```
CR      EQU  0Dh   ←—— 这一行不是真正的指令，不会
                        造成执行上的负担

.CODE
MOV     AL,CR          ；汇编后CR会被替换成0Dh
```

那么，我们是否可以用一个名称来代表一段程序呢？答案是可以的，在 MASM 中这项功能便叫做"宏"(MACRO)。宏有它自己的定义语法。

```
宏汇编名    MACRO
            ....  ┐
            ....  ├ 宏汇编体
            ....  ┘
            ENDM
```

每一个宏的定义，都要用 MACRO 和 ENDM 这两个伪指令来作为开始和结束。例如我们将 INT 21h 的结束程序功能定义成宏。

```
ENDPRG   MACRO
  ↑      MOV AH,4Ch    ;  ┐
  |      INT 21h       ;  ┘ 宏汇编的内容
宏汇编    ENDM
的内容
```

那么每次在程序中我们只要使用 ENDPRG 就可以结束程序了。

```
            .CODE
BEGIN:      ......
            .....
          ┌→ENDPRG ──────→在汇编时会替换成 ⎧ MOV  AH,4Ch
          │                                ⎩ INT  21h
      使用宏汇编
```

这么一来，我们就可以将一些常用的程序片段，以简短而有意义的名称来取代了，如此不仅使程序变得较为精简易读，而且也不会造成系统执行时的额外负担(因为在汇编时就将宏替换掉了)。

在汇编时，将宏名称以实际内容取代的步骤，我们就称为"宏展开"(Macro Expansion)。

以下范例是一个显示程序，我们每按一个键，程序便会在荧幕上再显示一个相同的字符，所以会出现双字的效果。如果按'!'则可以结束程序。

程序 PRG12-1.ASM

```
GETCHAR     MACRO                        ;读取字符的宏
            MOV AH,01H
            INT 21H
            ENDM
PUTCHAR     MACRO                        ;显示字符的宏
            MOV AH,02H
            INT 21H
            ENDM
ENDPRG      MACRO                        ;结束程序的宏
            MOV AH,4CH
            INT 21H
            ENDM

  ;————————程序由此开始————————
            .MODEL SMALL
            .CODE
BEGIN:      .WHILE 1
              GETCHAR                    ;使用宏读键
              .BREAK  .IF  AL=='!'       ;若输入'!'则离开
              MOV DL,AL                  ;否则把 AL 的键码存入 DL
              PUTCHAR                    ;并以宏显示出来
            .ENDW
            ENDPRG                       ;使用宏结束程序

            .STACK
            END  BEGIN
```

◆ 注：本程序不需要用到 DATA 段，所以.DATA 可省略。

实际上 MASM 是将程序做了以下的替换：

```
        BEGIN:          .WHILE  1                        ;使用MACRO输入字符
     *@C0001:
                       GETCHAR
     1                 MOV AH,01h
     1                 INT 21h

                        .BREAK .IF AL=='!'               ;若输入'!'则离开
     *            CMP   AL,'!'
     *            JE  @C0002
                        MOV  DL,AL                       ;否则把AL的ASCII码存入DL
                       PUTCHAR                           ;并以MACRO显示出来
     1                 MOV AH,02h
     1                 INT 21h
                        .ENDW
     *            JMP @C0001
     *@C0002:
                       ENDPRG                            ;使用MACRO结束程序
     1                 MOV AH,4Ch
     1                 INT 21h
                        .STACK
                       END BEGIN
```

◆ 注： 前端标有1的指令为宏汇编展开的结果，标有*的则为宏汇编伪指令的展开。

以上黑体的宏名都会被 MASM 替换成前面标有1的程序行。相较之下，使用 **GETCHAR**、**PUTCHAR**、**ENDPRG** 等宏，整个程序看起来更容易阅读，与子程序同样具有模块化、简洁、易懂的效果呢！

执行结果

```
C:\>PRG12-1
AABBCC@@##!
C:\>_
```

12.2 宏与子程序的比较

宏与子程序最大的不同，就在于宏只是汇编时的文字替换而已，而子程序则是执行时以 CALL 的方式来调用使用。以下将一一说明它们的差异，及适用的时机。

定义方式的差异

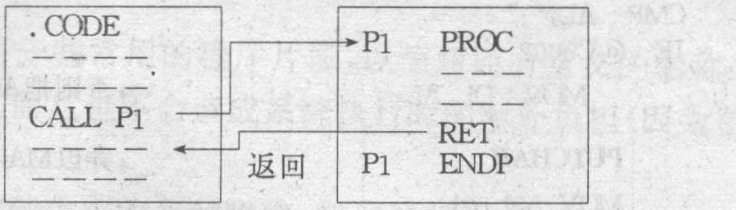
宏与子程序在定义上的差异如下：

子程序	宏汇编
P1　PROC 　　——— 　　RET ←—— 要RET P1　ENDP	M1　MACRO 　　——— 　　——— ←—— 无RET 　　ENDM ←—— 不加宏汇编名

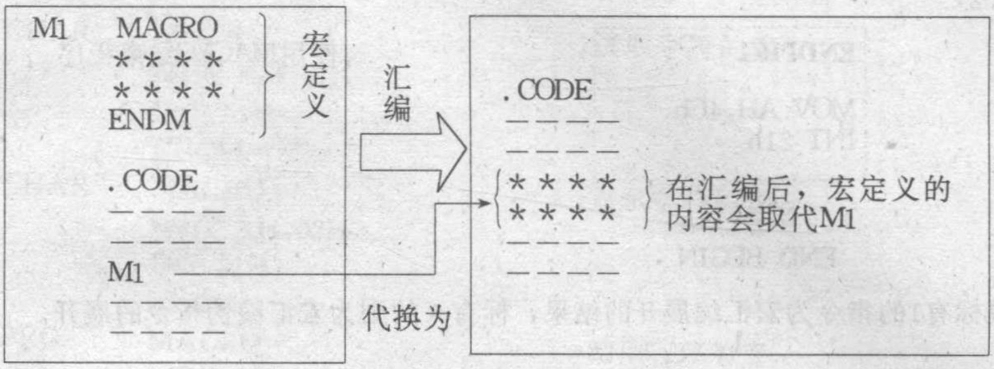
注意，宏的格式在 ENDM 的前面不需加上宏名称，而子程序则在 ENDP 之前要加上子程序名。

使用方式

子程序必须以 CALL 来调用，在子程序结束时，则要使用 RET 来返回原调用程序。例如：



宏在使用时只要直接写出宏名称即可，并不需 CALL 或 RET。然而，宏的定义必须出现在使用之前。



优缺点比较

由于子程序需要 CALL 与 RET，而宏则完全不必要，所以宏的执行速度较快。但是宏的展开事实上是程序的加入，每调用一次便加入一段程序，因而产生的执行文件会比较大。以下是二者的优缺点比较。

	子程序	宏汇编
调用方式	用CALL调用	使用宏名
返回方式	RET	无
执行速度	较慢	较快
执行文件	较小	较大

适用时机

由于子程序和宏都可以提高程序的可读性与可重用性，因此我们在编写程序时应多加利用。在比较过它们的优缺点后，我们可以得到以下的结论：

1)若程序码很短，或使用次数很多，则使用宏较佳。

2)若程序码很长，或使用次数很多，则使用子程序较佳。

3)若我们希望程序文件能小一点，则尽量使用子程序。

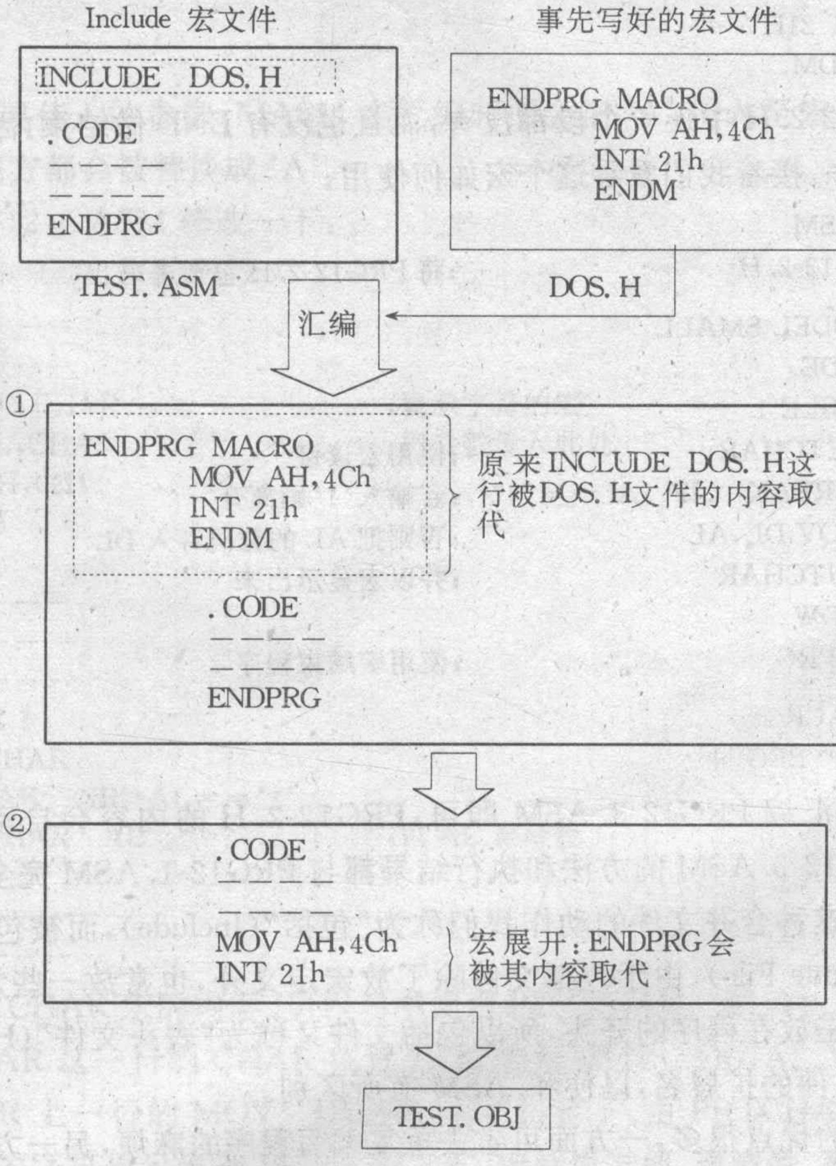
4)若我们希望程序执行快一点，则尽量使用宏。

以上只是一些建议，读者可视方便性及实际情况来决定要使用那一种方式。

12.3 INCLUDE 伪指令

编写好的宏就像子程序一样，可以一再重复使用。然而，子程序可以分别存放在不同的文件中，并汇编成 OBJ 文件或放入程序库中以供其他程序连接(Link)成执行文件；但宏却无法这样做，由于它在汇编时就要做宏展开，所以宏的定义一定要出现在每一个使用它的程序文件中，如此 MASM 才知道要如何来做替换。

INCLUDE 伪指令就是用来简化使用宏时的不便，我们可以先将各种宏存放在独立的文件中，然后再用 INCLUDE 来将编写好的宏文件拷贝进我们程序文件中。例如：



在汇编时，MASM 会先看看程序文件中有没有 INCLUDE 指令，若有，则把 INCLUDE 指定的文件内容包括(Include)到程序里头(如上例的 DOS.H)。处理完所有的 INCLUDE 指令后，MASM 才会继续进行其他的汇编工作。

在做包括的动作时，包括进来的文件会取代 INCLUDE 指令而插入到 INCLUDE 所在的位置。我们可以把常用的常数定义、宏等写成一个文件，然后在每个需要用到这些定义、宏的程序中，加上一行"INCLUDE 文件名"，这样 MASM 便会先把指定的文件插入到程序中对应

的位置，然后再进行汇编。

以下我们将 PRG12-1. ASM 中的宏定义独立出来。

程序 PRG12-2. H

```
GETCHAR     MACRO                          ;读取字符的宏
            MOV AH,01H
            INT 21H
            ENDM
PUTCHAR     MACRO                          ;显示字符的宏
            MOV AH,02H
            INT 21H
            ENDM
ENDPRG      MACRO                          ;结束程序的宏
            MOV AH,4CH
            INT 21H
            ENDM
```

请注意，PRG12-2. H 中连一个段都没有，而且也没有 END 做结束！因为这只是存放宏的文件，不是一般程序。接着我们看看这个宏如何使用。

程序 PRG12-3. ASM

```
INCLUDE     PRG12-2. H                     ;将 PRG12-2. H 包括进来

            . MODEL SMALL
            . CODE
BEGIN:      . WHILE 1
              GETCHAR                      ;使用宏读键
              . BREAK  . IF  AL==’!’       ;若输入’!’则离开
              MOV DL,AL                    ;否则把 AL 的键码存入 DL
              PUTCHAR                      ;并以宏显示出来
            . ENDW
            ENDPRG                         ;使用宏结束程序

            . STACK
            END  BEGIN
```

我们只要直接汇编 PRG12-3. ASM 即可，PRG12-2. H 的内容会自动包括进 PRG12-3. ASM 中。汇编 PRG12-3. ASM 的方法和执行结果都与 PRG12-1. ASM 完全相同。

像 INCLUDE 这种合并文件的动作我们称为“包括”(Include)，而被包括进来的文件则叫做“包括文件”(Include File)，由于包括文件除了放宏定义外，也常放一些常数定义，而这些宏及常数定义通常都是放在程序的开头，所以包括文件又称为“表头文件”(Header File)。我们通常会以. H 为表头文件的扩展名，以便和. ASM 有所区别。

使用包括文件的优点很多，一方面可省去重复编写程序的麻烦，另一方面可便利程序的维护。就像在上例中建立 PRG12-2. H 便可以一用再用，只要在程序内指定″INCLUDE PRG12-2. H″，则汇编时 PRG12-2. H 便会被合并到程序内。若发现 PRG12-2. H 中的宏有错误，也是需修改 PRG12-2. H 的内容，而不必改动其他文件。

12.4 宏的参数传递

我们可以在宏定义中放入参数，例如前例中的 PUTCHAR 宏可以加以改良：

```
                  MACRO定义
                     ↓          ┌──────────── 参数
                                ↓
PUTCHAR        MACRO CHAR ──────┘
               MOV DL,CHAR ←──────────── 代入此处 ←
               MOV AH,02h
               INT 21h
               ENDM
```

上列宏定义后面的 char 就是一个参数，假设在程序中有这样一行宏指令：

```
PUTCHAR 'A'   ⇨   MOV DL,'A'
                  MOV AH,02h
                  INT 21h
```

则 'A' 就是我们要传入的参数，MASM 在汇编时便会把 'A' 传入至宏内对应的位置，凡是宏内有 CHAR 的地方都会被替换成 'A'。

现在把程序 PRG12-1. ASM 修改一下：

程序 PRG12-4. ASM（只列出修改的部分）

```
            ———————
PUTCHAR     MACRO   CHAR                    ;显示字符的宏
            MOV DL,CHAR                     ;把参数传入此处
            MOV AH,02H
            INT 21H
            ENDM
            ———————
            ———————
            ———————
BEGIN:      .WHILE 1
              GETCHAR
              .BREAK  .IF  AL=='!'
              PUTCHAR   AL                  ;以 AL 为参数
            .ENDW
            ———————
            ———————
```

我们修改了 PUTCHAR 宏的内容，当程序中使用 PUTCHAR　AL 时，AL 便会传入到宏内的 MOV　DL,CHAR 这一行的 CHAR 位置上，而成为 MOV　DL,AL。原来程序 PRG12-1. ASM 中 PUTCHAR 上一行的 MOV　DL,AL 就不必要了。由 PRTCHAR　AL 可以很明白的知道是要把 AL 的字符显示出来，使用这种方式，宏就更方便、更清楚、更具完整性了。

宏的参数可以是任何的常数数据、寄存器、或变量（数据标号），反正宏参数所做的是替换工作（参数的加入），把握此原则就对了。下面例子示范如何使用常数和变量来当作参数。

程序 PRG12-5. ASM

```
PUTCHAR        MACRO   CHAR
               MOV     DL,CHAR
               MOV     AH,02H
               INT     21H
               ENDM
```

```
          .8086
          .MODEL SMALL
          .CODE
CHA       DB        'B'
          .CODE
BEGIN:    MOV    DX,@DATA
          MOV    DS,DX
          PUTCHAR   'A'               ;显示出'A'字符
          PUTCHAR   CHA               ;显示出 CHA 变量('B'字符
          MOV    AH,4CH
          INT    21H
          .STACK
          END    BEGIN
```

上述程序中，我们分别以常数'A'和变量 CHR 当做 PUTCHAR 的参数传入宏中，所以程序最后会显示出"AB"。

12.5 宏内的标号与 LOCAL 定义

在宏的内部，我们也可以使用程序标号来做流程的控制，但由于一个宏可能会被调用多次，那么在宏展开后，宏中的标号不就会重复好几次了吗?为了防止这样的问题，在宏中的标号必须先以 LOCAL 伪指令加以定义。

例如下面的程序，我们在 PUTCHS 宏中加上一个程序标号，并多加了一个新的参数 TIMES 来控制重复显示字符的次数。

程序 PRG12-6.H

```
                                      将宏汇编中的标号以LOCAL定义
                                                   |
PUTCHS   MACRO CHAR,TIMES             ;使用两个参数  ↓
         LOCAL REP1                   ;定义REP1为局部使用
         MOV CX,TIMES
REP1:    MOV DL,CHAR                  ;  }
         MOV AH,02h                   ;  }
         INT 21h                      ;  } 执行TIMES次
         LOOP REP1                    ;  }
         ENDM
```

REP1经 LOCAL 定义后，每次在宏展开时，MASM 都会将 REP1改为另外一个不会重复的编码，因此每个宏展开中的标号都不可能会重复了。接着，我们以下面的程序来调用此宏。

程序 PRG12-7. ASM

```
    INCLUDE      PRG12-6.H                        ;将 PRG12-6.H 包括进来
                 .8086
                 .MODEL SMALL
                 .CODE
    BEGIN:       PUTCHS  'A',3                    ;显示3个'A'
                 PUTCHS  'B',5                    ;显示5个'B'
                 MOV   AH,4CH
                 INT  21H
                 .DATA
                 .STACK
                 END  BEGIN
```

程序执行后正确地显示出了:'AAABBBBB'

12.6 程序举例

在本章例中,我们将各种常用的 INT 21h 功能收录于 DOS.H 中,然后修改第八章的实例 PRG8-1.ASM 来使用 DOS.H。

12.6.1 DOS.H 文件

程序 DOS.H 常用的 DOS 宏

```
;-----------------------------------------
;DOS  功能调用  01H:读取按键并显示到屏幕上
;返回:(CHAR)=(AL)=键入字符的 ASCII 码
;-----------------------------------------
GETCHAR     MACRO  CHAR
            MOV   AH,01H
            INT   21H
            MOV   CHAR,AL
            ENDM
;-----------------------------------------
;DOS  功能调用  02H:显示字符到屏幕上
;-----------------------------------------
PUTCHAR     MACRO  CHAR
            MOV   DL,CHAR
            MOV   AH,02H
            INT   21H
            ENDM
;-----------------------------------------
;DOS  功能调用  06H:读取按键但不回显不等候
;返回:Z=0  (AL)=键入字符的 ASCII 码,Z=1  无按键
;-----------------------------------------
GETC     MACRO
         MOV  DL,0FFH
         MOV  AH,06H
         INT  21H
         ENDM
;-----------------------------------------
;DOS  功能调用  09H:显示一字符串到屏幕上
```

```
;——————————————————————————————
PRINTSTR    MACRO  STRING
            MOV  DX,OFFSET  STRING
            MOV  AH,09H
            INT   21H
            ENDM
;——————————————————————————————
;DOS  功能调用  0AH:读取一字符串
;STRING 的前2字节分别为最大输入长度,
;                以及输入字符串的长度
;——————————————————————————————
INPUTSTR    MACRO  STRING
            MOV  DX,OFFSET  STRING
            MOV  AH,0AH
            INT   21H
            ENDM
;——————————————————————————————
;DOS  功能调用  4CH:结束程序
;——————————————————————————————
ENDPRG      MACRO
            MOV  AX,4CH
            INT  21H
            ENDM
```

12.6.2 PRG12-9.ASM

程序 PRG12-9.ASM

```
INCLUDE     DOS.H
            .286                    ;设定汇编成286以上都适用的机器码
            .MODEL  SMALL           ;设定为 SMALL 存储器模式

            .DATA
BUF         DB   80,0,80 DUP(?)     ;定义82字节来存放字符串
                                    ;但最多只能存放80个字符
            .CODE
BEGIN:      MOV  AX,@DATA
            MOV  DS,AX              ;将 DS 指向 DATA 段
            INPUTSTR  BUF           ;调用读取字符串的宏

            MOV  CH,0               ; }
            MOV  CL,BUF+1           ; } CX 存放输入的字符数目
            MOV  BX,2               ;BX 作为地址指针,由字符串的
                                    ;第两个字节开始处理

CHKBUF:     .IF  BUF[BX]==''        ;检查是否为空白字符←┐
              MOV  BUF[BX],'_'      ;若是,则将''换成'_'  │
            .ENDIF                                      │
            INC  BX                 ;                   │
            LOOP  CHKBUF            ;循环(执行 CX 次)────┘
            MOV  BUF,13             ;将 BUF 最前面的二字节
            MOV  BUF+1,10           ;     换成 CR 及 LF
            MOV  BUF[BX],'$'
            PRINTSTR  BUF           ;调用显示字符串的宏

            ENDPRG                  调用结束程序的宏
```

```
        .STACK
        END  BEGIN                    ;程序结束
```

执行结果

```
C:\>PRG12-9
ABC      sdf  df  dffff  [Enter]
ABC __ sdf _ df _ dffff
```

12.6.3 更进一步的模块化

在上面的实例中,我们应可发现程序中有许多功能以后可能会经常用到,例如读取指定长度的字符串、将字符中的某字符换成另一个字符、显示出指定长度的字符串(而不需以'$'做为结尾)、及换行操作等,都可以将之编写成宏或子程序,以提高程序的可重用性。

以下我们修改 PRG12-9.ASM,将其中常用的功能也独立出来成为宏,并存入 UTIL.H 中。

程序 UTIL.H

```
    INCLUDE     DOS.H        ;由于会使用到 DOS.H 中的宏,所以要包括 DOS.H
;———————————————————————————————————————
;INITDS:将 DS 指向 DATA 段
;———————————————————————————————————————
INITDS      MACRO            ;将 DS 指向 DATA 段
            MOV  AX,@DATA
            MOV  DS,AX
            ENDM
;———————————————————————————————————————
;READSTR:读取字符串,MAXLEN 为最大输入长度,
;        输入的长度将存于 READLEN 中
;———————————————————————————————————————
READSTR   MACRO   STR,MAXLEN,READLEN
          MOV   STR,MAXLEN
          INPUTSTR   STR
          MOV   AL,STR[1]       ;避免 MOV   mem,mem
          MOV   READLEN,AL      ;       的不合法情形
          ENDM
;———————————————————————————————————————
;REPLSTR:字符串中的字符替换,由第 START 个字符开始
;        检查 LEN 个字符,将其中的 OLD 换成 NEW
;———————————————————————————————————————
REPLSTR   MACRO   STR,START,LEN,OLD,NEW
          LOCAL   L1
          MOV   CH,0
          MOV   CL,LEN               ;CX 储存要检查字符的个数
          MOV   BX,START             ;BX 指向字符串的第 START 字符
   L1:    .IF   BUF[BX]==OLD         ;当检查到 OLD 字符←
            MOV   BUF[BX],NEW        ;则以 NEW 字符替换
          .ENDIF                     ;
          INC   BX                   ;指向下一个字符
          LOOP  L1                   ;执行 CX 次——
          ENDM
```

```
;---------------------------------------
;WRITESTR:显示出长度为 LEN 的字符串
;---------------------------------------
WRITESTR  MACRO  STR,LEN
          MOV  BH,0
          MOV  BL,LEN            ;BX 指向字符串结束地址
          MOV  STR[BX],'$'       ;储存字符串结束字符
          PRINTSTR  STR          ;显示字符串
          ENDM
;---------------------------------------
;LINEFEED:输出换行字符 CR 及 LF
;---------------------------------------
LINEFEED  MACRO
          PUTCHAR  13            ;显示 Line Feed
          PUTCHAR  10            ;显示 Carriage Return
          ENDM
```

程序 PRG12-11. ASM

```
INCLUDE   DOS. H
INCLUDE   UTIL. H
          .286                   ;设定汇编成286以上都适用的机器码
          .MODEL  SMALL          ;设定为 SMALL 存储器模式

          .DATA
BUF       DB    82 DUP(?)        ;定义82字节来存放字符串
LEN       DB    ?                ;用来储存字符串的长度

          .CODE
BEGIN:    INITDS                        ;将 DS 指向 DATA 段
          READSTR  BUF,80,LEN           ;读取字符串
          REPLSTR  BUF,2,LEN,' ','_'    ;替换字符(由第2个字节开始)
          LINEFEED                      ;换行
          WRITESTR  BUF+2,LEN           ;由 BUF+2处显示出 LEN 字符
          ENDPRG                        结束程序
          .STACK
          END  BEGIN
```

经过更进一步的模块化之后,我们的主程序变得很简洁,而且 UTIL. H 中的宏还可供其他的程序文件使用,这就是模块化的好处。

执行结果

```
C:\>PRG12-11
This is the final chapter. [Enter] ←——输入此字符串
This_is_the_final_chapter.
```

本章习题

1. 请解释下列名词?

   宏 (Macro)

   宏展开 (Macro Expansion)

   包括 (Include)

   包括文件 (Include File)

表头文件（Header File）

2. 请分析宏和子程序的优缺点，并说明二者的适用时机。

3. 请说明 INCLUDE 伪指令的作用与原理。

4. 请详述宏的参数传递方式，并举例说明。

5. 在宏中使用参数时，若有下列情形：

```
COPY        MACRO  P1,P2
            MOV    P1,P2
            ——————
            ENDM

            .CODE
            ——————
            COPY   AX,D2        ;正确
            COPY   D1,D2        ;错误
            ——————
```

在宏展开后，由于 MOV 的两个操作数不可同时为存储器数据，因此 MOV D1,D2会 造成错误。请问要如何避免此一问题？

6. 在宏中何时须使用 LOCAL 伪指令，其功能为何？

7. 请利用本章实例中的 DOS. H 或 UTIL. H 编写一个程序，可以显示出下面文字：

"经过不断的努力、拼打…"

"我终于学会汇编语言了，哈哈！"

8. 请编写一个可以将字符串内容反转的宏（例如"ABC" "CBA"），以 REVSTR 为名并加入 UTIL. H 中。

9. 试编写一个程序可以由键盘读入字符串，并将字符串内容反转后显示出。

10. 请编写一个计算字符串长度的宏（字符串以 ASCII 0 作为结尾），以 STRLEN 为名并加入 UTIL. H 中。

11. 请将本书介绍过的 BIOS 调用编写成宏，并存入 BIOS. H 中，并测试其正确性。

12. 请扩充 DOS. H 文件，将所有您认为有用的功能调用都加入此文件中，并测试其正确性。

13. 请编写一个猜数字游戏，程序流程如下：

    ①读取系统时间，并利用分、秒数产生一个随机的四位数（必须四个数字都不相同），以作为要猜的答案。

    ②进入一个循环，不断由键盘读取使用者输入的四位数字，然后用以下规则显示猜的情形：

    □若数字相同但所在位置不相同，则以一个 B 表示。

    □若数字相同且所在位置也相同，则以一个 A 表示。

    □当结果为4A（猜中了）时则离开循环。

    例如答案为2501，使用者猜0532，则回应1A2B（5：位置相同所以为 A，02：位置不同所以为 B）。

    ③将使用者一共猜了多少次显示出来。

14. 改良上一题，加上时间计算功能，若超过一小时则自动结束程序，此外，在结束前也要将一共花了多少时间显示出。

附录A　ML. EXE　的命令参数

ML. EXE 可以汇编和连接一个或多个汇编语言源程序，其使用格式如下：

ML　[控制参数]　文件名　[[控制参数]文件名]....

图 B.1　ML　的格式

注意，在控制参数中的大写字母和小写字母所代表的意义是不同的，因此我们必须依照其指定的大小写格式来给定命令。

由于 ML 可以汇编、连接多个程序，因此其控制参数是分别加在每个文件名称的前面，例如：

ML　/Fm PGM5—1. ASM　/F1　/Sc　PGM5—2. ASM

上例中的 /Fm 参数仅对 PGM4—0. ASM 有效，而 /F1　/Sc 则仅对 PGM4—1. ASM 有效。

ML 的命令参数共有46个，以下仅列出其中较常使用的参数：

□/?：列出 ML 的所有参数。

□/help：启动 ML 的帮助功能，提供所有 MASM 的详细说明。

□/AT：指定以 Tiny 存储器模式汇编程序，或程序并非 Tiny 的存储器模式时，汇编时会显示错误讯息。

□/B1连接器文件名：指定用 LINK. EXE 以外的连接器来连接，例如使用/B1TLINK. EXE 来指定使用 Borland 的 TLINK. EXE 连接器。

□/c：只汇编而不连接。

□/Cp：在汇编时，保留程序中代名的大小写，所以程序中大小写不一样的代名将会被视为不同的代名，例如："PutStr"与"PUTSTR"会被视为不同的代名。

□/Cu：先将程序中的代名全部转换成大写后，再进行汇编、连接，所以程序中的"PutStr"与"PUTSTR"将会被视为相同。

□/Cx：对程序中定义为外部(External)或公用(Public)的变量、子程序名、段名，保留其原来的大小写，其余均转换为大写后，再行汇编、连接(这是缺省的汇编方式)。

□/Fe 文件名：指定经汇编、连接后，所产生的 EXE 文件的文件名。

□/F1文件名：产生 LST 文件，并可以指定其文件名。

□/Fm 文件名：产生 MAP 文件，并可以指定其文件名。

□/Fo 文件名：指定经汇编后，所产生的 OBJ 文件名。

□/Fpi：对于一些没有协处理器(例如：80387)的电脑，当在程序中使用了浮点运算指令时，则必须使用此参数来汇编，模拟浮点运算指令(FloatingPoint Emulator)的功能。

□/I 路径名称：指定 INCLUDE 伪指令中的. H 文件和. INC 文件所在的路径，最多可有10个/I 参数。

□/nologo：若汇编成功无误，则不显示任何讯息。

□/Sc：在 LST 文件中列出每一个指令的执行时间。

□/Sf：在 LST 文件中列出汇编所产生的错误讯息，以及展开所有的宏和流程控制伪指令。

□/S1字数：在列出 LST 文件时，指定每行所列出的字数，字数的范围只能是60～255或0，缺省值为0，代表不做限制。

□/Sn：在列出 LST 文件时，不列出符号(Symbol Table)表。符号表是一种将代名及存储地址对照列出的表格，通常是附加在 LST 文件后面的位置。

□/Sp 行数：在列出 LST 文件时，指定每页所列出的行数。行数的范围只能是10～255或0，缺省值为0，代表不做任何分页。

□/St 字符串：指定 LST 文件中的标题(Title)字符串，这个字符串会在每页开头的地方显示。

□/Ss 字符串：指定 LST 文件中的副标题(Subtitle)字符串，这个字符串会从第2页开始显示，并且接在标题的后面。

□/Ta 文件名：汇编不是以.ASM 为扩展名的汇编语言程序。

□/Zf：将所有的代名定义为 Public。

□/Zi：把程序行号及变量名、型别列入到 OBJ 文件中，以供 Code View 侦错用。

□/Zd：仅将程序行号加入 OBJ 文件中，以供 Code View 侦错用。

□/Zm：指明要汇编 MASM5.1版的程序，由于在5.1版中有些特殊语法不适用于6.x 版，如果我们的程序中有用到这些语法，则必须加/Zm 参数来汇编。

□设定系统配备文件：NEW—CONF.SYS 文件中则包含了系统配备所需的命令，假设原先我们的 CONFIG.SYS 中已经有了这些命令，请修改其中的参数以符合在 NEW—CONF.SYS 中的最低要求。其内容如下：

```
files=20
buffers=10
device=C:\MASM611\BIN\himem.sys
device=C:\MASM611\BIN\emm386.exe   8192   RAM
REM   Use the follow ing line to run CodeView without Windows or 386MAX loaded.
REM   device=C:\MASM611\BIN\emm386.exe   2048   RAM
```

要特别注意的是，要在 DOS 下执行 CV.EXE(Code View)时，在 CONFIG.SYS 文件中的 EMM386.EXE 驱动程序不可以使用/NOEMS 参数，否则会造成程序无法执行的问题，而且在执行 LINK.EXE 文件时，也会出现以下警告讯息：

"Warning running program in real mode: CPU already in
protected or virtual mode"

若读者要在 DOS 下使用 CV.EXE，或在执行 LINK.EXE 时不希望看到以上警告信息则可以将 CONFIG.SYS 文件该行指令改为：

DEVICE=C:\MASM611\BIN\EMM386.EXE RAM 2048

↖—由于 MASM 611所提供的 EMM386.EXE 版本较 DOS 6.2x 所提供的驱动程序旧，因此我们可以将此段改为 DOS 所在的目录，以使用较新的版本。

在完成上述的系统设定之后，请重新开机使电脑设定新的系统环境，如此才能使 MASM 正确的工作。

附录B　MASM 6.11的系统需求与安装程序

系统需求

以下为执行 MASM 6.11的最低系统需求：

□80386以上的兼容主机及4M RAM。

□MS－DOS3.3以上的操作系统。

□所需硬盘空间至少为10MB。

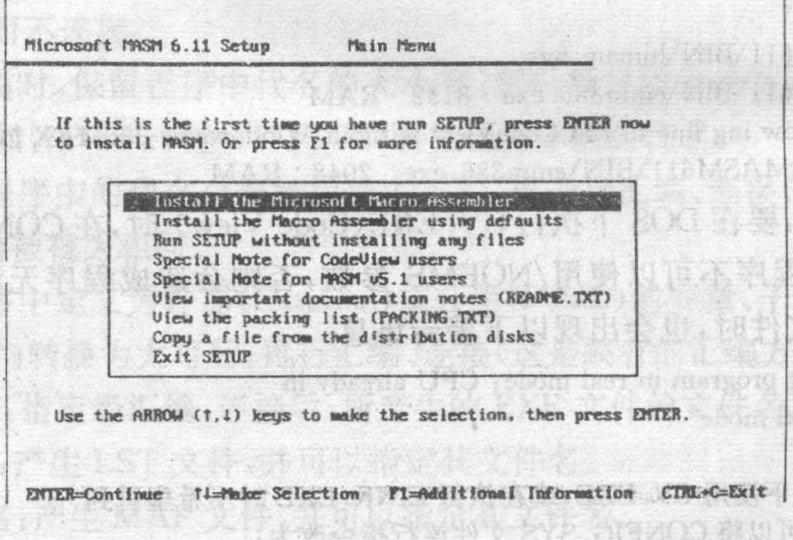
安装　MASM

在安装 MASM 之前，建议您先以 DOS 的 DISKCOPY 命令制作备份，然后用备份的磁盘来做安装的工作。MASM6.11版安装磁盘中的文件都是经过压缩的，因此必须使用安装程序才能安装到硬盘上。以下为 MASM6.11的安装步骤：

1. 将 Disk 1磁盘放入驱动器中。

2. 将目前的驱动器切换至上述磁盘所在的驱动器。

3. 在 DOS 下键入 SETUP [Enter]，接着在第一个画面出现后再按[Enter]，屏幕便会显示 SETUP 的主画面。



安装程序的主菜单画面

4. 选择 **Install theMicrosoft Macro Assenbler** 后按[Enter]键，此时屏幕会依序出现以下选项：选择操作系统环境、是否安装 Windows 程序、PWB 集成开发环境、将 PWB 模拟成 Brief 的文书编辑器、鼠标驱动程序、MASM.EXE、帮助说明文件、范例程序、以及设定各类文件所要安装的目录。我们可根据实际需求设定安装内容。若对选项内容不清楚时可使用[F1]键来查阅帮助说明，确定后再按[Enter]键，直到 SETUP 的确认安装画面出现。

```
Microsoft MASM 6.11 Setup          Confirm Your Choices

   If you are satisfied with your responses, press ENTER to continue.
   Otherwise, use the ARROW (↑,↓) keys to select the option to change
   and press ENTER. Your current response is the new default.

   NO CHANGES
   Install for MS-DOS/Microsoft Windows
   Install Microsoft Windows tools
   Install Microsoft Programmer's WorkBench
   Do not install Brief compatibility
   Copy Microsoft Mouse software
   Install the MASM 5.1-compatible driver
   Copy documentation files
   Copy sample programs
   Directory for executable files:  C:\MASM611\BIN\
   Directory for library files:  C:\MASM611\LIB\
   Directory for initialization files:  C:\MASM611\INIT\
   Directory for include files:  C:\MASM611\INCLUDE\
   Directory for help files:  C:\MASM611\HELP\
   Directory for sample files:  C:\MASM611\SAMPLES\
  -MORE-
ENTER=Continue    ↑↓=Make Selection                          CTRL+C=Exit
```

安装程序的确认安装项目画面

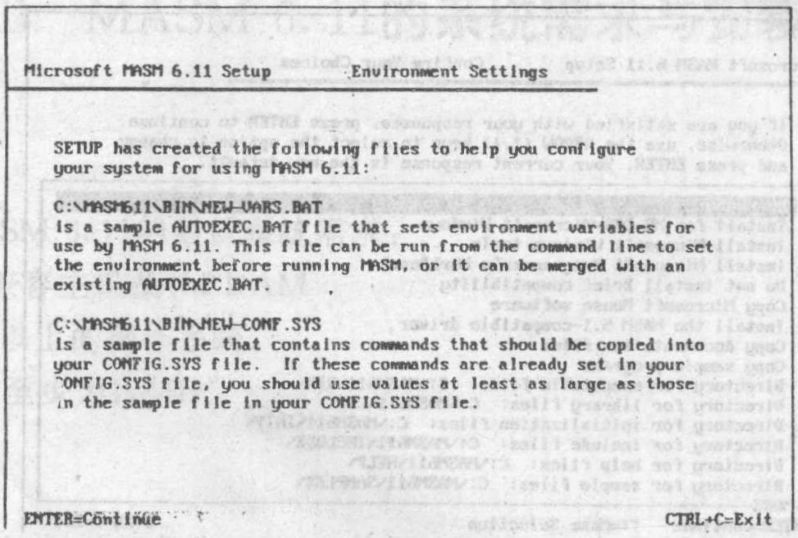
5. 此时，我们可对前面的各项设定做最后的检查及修改。在确定后可选择 NOCHANGES 然后按[Enter]，程序便会开始将文件安装至硬盘上。如果我们的硬盘容量不够，则 SETUP 程序会提出警告并请我们适当的处理。

6. 接着我们只要依照指示更换磁盘，直到 Installation Complete 画面出现。

```
Microsoft MASM 6.11 Setup          Installation Complete

   SETUP has successfully installed MASM 6.11.

   In addition to copying MASM 6.11 onto your hard drive,
   SETUP has created several new files.  The following
   screens will tell you how to use these files.

   There are differences between versions 5.1 and 6.11.
   If you will be assembling code written for MASM 5.1,
   please choose the "Special Note for MASM v5.1 users"
   from the SETUP main menu.  Additional information is
   available in Appendix A of the Programmer's Guide.

   Press ENTER to continue, or press CTRL+C to exit SETUP.

ENTER=Continue                                       CTRL+C=Exit
```

安装完毕画面

7. 紧接着安装程序会依序显示如何设定系统环境、SYSTEM.INI 文件、PWB 环境、存储器管理公用程序等说明画面，在读完说明后可以按[Enter]键以进入下一个画面。有关系统的设定我们会在稍后介绍。

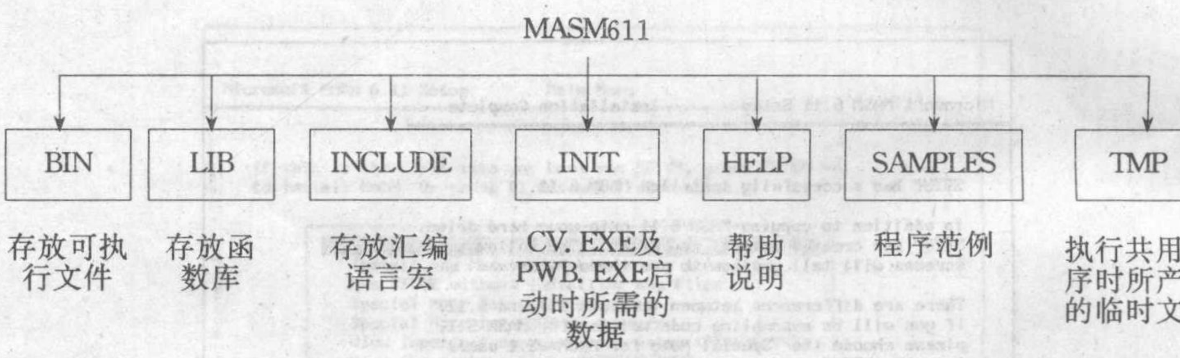


```
Microsoft MASM 6.11 Setup          Environment Settings

SETUP has created the following files to help you configure
your system for using MASM 6.11:

C:\MASM611\BIN\NEW-VARS.BAT
is a sample AUTOEXEC.BAT file that sets environment variables for
use by MASM 6.11. This file can be run from the command line to set
the environment before running MASM, or it can be merged with an
existing AUTOEXEC.BAT.

C:\MASM611\BIN\NEW-CONF.SYS
is a sample file that contains commands that should be copied into
your CONFIG.SYS file.  If these commands are already set in your
CONFIG.SYS file, you should use values at least as large as those
in the sample file in your CONFIG.SYS file.

ENTER=Continue                                          CTRL+C=Exit
```

环境设定画面

8. 在一连串系统设定的说明画面结束后,安装程序会再度回到安装主画面,我们可以选择 Exit SETUP 选项,安装程序便大功告成。

在安装时,若没有更改任何缺省的目录,则所有的文件会分别被安装在 MASM611目录下的七个子目录中,如下图所示:



MASM 6.11 主要的目录结构

其中在 BIN 的子目录下会有下列几个较重要的执行文件:

□ML. EXE 及 LINK. EXE:用来汇编源程序和连接目的程序。

□PWB. EXE:是一个汇编语言集成开发环境(Integrated Environment),提供编辑、汇编、连接、除错、及执行程序等功能,使用者可在此环境下完成整个程序的开发工作。

□CV. EXE:针对 Microsoft 的各种语言做源程序的侦错,尤其是在连接高阶语言及汇编语言的程序时,是不可缺少的工具。

系统的设定

前面所提到的 SETUP 程序并未对我们的系统作任何修改,因此,系统可能还无法正常地工作。当安装完成后,SETUP 程序会提供有关系统设定的建议,我们可以依照这些说明对系

统作一些修改：

□设定自动执行文件：在 MASM611\BIN\目录下的 NEW—VARS. BAT 文件中设定了一些 MASM6. 11会使用到的环境变量，我们可在使用 MASM 之前执行此批处理，或直接将文件内容合并至 AUTOEXEC. BAT 批处理文件中。其内容如下：

```
SET PATH=C:\MASM611\BIN;%PATH%
SET LIB=C:\MASM611\LIB
SET INCLUDE=C:\MASM611\INCLUDE;%INCLUDE%
SET INIT=C:\MASM611\INIT;%INIT%
SET HELPFILES=C:\MASM611\HELP\*.HLP;%HELPFILES%
SET ASMEX=C:MASM611\SAMPLES;%ASMEX%
SET TMP=C:\MASM611\TMP
```

Document generated by Anna’s Archive around 2023-2024 as part of the DuXiu collection (https://annas-blog.org/duxiu-exclusive.html).

Images have been losslessly embedded. Information about the original file can be found in PDF attachments. Some stats (more in the PDF attachments):
{
"filename": "MTM4Mjk3ODAuemlw",
"filename_decoded": "13829780.zip",
"filesize": 127722996,
"md5": "7b5dd865e26b0800b3aaff8cce1bb34d",
"header_md5": "c584870dd64cb29c56208e8d2c66a97b",
"sha1": "d371304b5dcbb80752ccf0a775af2e5760afd325",
"sha256": "82ffb15117e66880fcedca695e38fbac3f3224bebe8651394740c48d79076cbe",
"crc32": 2603740460,
"zip_password": "52gv",
"uncompressed_size": 129417640,
"pdg_dir_name": "DOS 6.22\u6280\u672f\u624b\u518c \u6c47\u7f16\u8bed\u8a00\u6559\u7a0b_13829780",
"pdg_main_pages_found": 199,
"pdg_main_pages_max": 199,
"total_pages": 206,
"total_pixels": 1175578080,
"pdf_generation_missing_pages": false
}